University of the Witwatersrand

School of Computer Science and Applied Mathematics

Faculty of Science

Masters Dissertation

A Comparative Analysis of Dynamic Averaging Techniques in Federated Learning

Sashlin Reddy

Supervisor(s): Prof. Turgay Celik and Dr. Hairong Wang

October 2020, Johannesburg

Declaration

University of the Witwatersrand, Johannesburg

School of Computer Science and Applied Mathematics

SENATE PLAGIARISM POLICY

I, SASHLIN REDDY, (Student number: 675723) am a student registered for COMS8003A in the year 2020.

I hereby declare the following:

- I am aware that plagiarism (the use of someone else's work without their permission and/or without acknowledging the original source) is wrong.
- I confirm that ALL the work submitted for assessment for the above course is my own unaided work except where I have explicitly indicated otherwise.
- I have followed the required conventions in referencing the thoughts and ideas of others.
- I understand that the University of the Witwatersrand may take disciplinary action against me if there is a belief that this in not my own unaided work or that I have failed to acknowledge the source of the ideas or words in my writing.

Signature: 

Signed on 25th day of October , 2020 in Johannesburg.

Abstract

Due to the advancements in mobile technology and user privacy concerns, federated learning has emerged as a popular machine learning (ML) method to push training of statistical models to the edge. Federated learning involves training a shared model under the coordination of a centralized server from a federation of participating clients. In practice federated learning methods have to overcome large network delays and bandwidth limits. To overcome the communication bottlenecks, recent works propose methods to reduce the communication frequency that have negligible impact on model accuracy also defined as model performance. Naive methods reduce the number of communication rounds in order to reduce the communication frequency. However, it is possible to invest communication more efficiently through dynamic communication protocols. This is deemed as **dynamic averaging**. Few have addressed such protocols. More so, few works base this dynamic averaging protocol on the diversity of the data and the loss. In this work, we introduce dynamic averaging frameworks based on the diversity of the data as well as the loss encountered by each client. This overcomes the assumption that each client participates equally and addresses the properties of federated learning. Results show that the overall communication overhead is reduced with negligible decrease in accuracy.

Acknowledgements

The author would like to thank Prof. Turgay Celik and Dr. Hairong Wang for their reviews and suggestions on the overall system design. Prof. Turgay Celik and Dr. Hairong Wang always made themselves available whenever the author needed to exchange views on ideas.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

A wealth of data is generated daily by billions of mobile phones and internet of things (IoT) devices. Big data holds significant potential for various reasons and machine learning has flourished as a tool to harvest this data. In particular, large-scale machine learning has increasingly become an important phenomenon with the availability of big data. Such data has gone far beyond the capacity of a single machine for optimization problems defined in Section 2.1. To solve big data optimization problems, we require distributed algorithms that rely on inter-machine communication (Zhang & Xiao, 2015).

For optimization problems, gradient optimization has been a commonly used method due to it's efficiency and effectiveness. This also expands to distributed optimization. Stochastic gradient descent (SGD) has largely been the de factor gradient optimization method due to it's simplicity. It has been presented in a distributed setting under the name *parallel-SGD* (Stich, 2018). As with many algorithms in a distributed setting, communication overhead is a significant problem. This has received significant attention. Overall, these methods have typically worked well when data is stored on a centralized server. We refer to this as centralized data.

Over recent times, there has been large concerns over user privacy (Sahu et al., 2018). There have been improvements in user privacy policies but centralized data collection remains ripe for privacy risks (Shokri & Shmatikov, 2015). Moreover, in domains such as medicine, it is not permitted by law to share data about individuals. Thus, the concept of decentralized data has emerged to ensure data privacy. Unlike centralized data, decentralized data is not stored on a centralized server or managed by a centralized authority. Rather it is stored on the end user's device or decentralized data centers. In the case of the medicine domain example, data is stored in the data center of a health institution. Since we are not sharing data with a centralized server, there is less risk from a privacy perspective. Consequently, this changes large-scale machine learning methods. Fortunately, with the advancements of technology, it is possible to perform machine learning on decentralized data. Federated learning (Brendan McMahan and Daniel Ramage, 2017) has emerged as a popular

machine learning method to build models on decentralized data. Federated learning is a method that involves training a shared machine learning model from a federation of participating devices on decentralized data. This process is coordinated by a centralized server. Unlike traditional large-scale machine learning methods, participating devices do not share data with a centralized server. They instead perform a local learning process and communicate model updates to the centralized server. The centralized server aggregates the updates received from these devices and sends the updated model updates to all participating devices. This process is repeated for a selected number of iterations. One of the benefits from a privacy perspective is that these model updates do not need to be stored. This is since they generally contain less information than the raw data used for the learning process. Federated learning also allows domains that have strict regulations to learn shared models. This may accelerate research in these domains since models are learning over larger datasets.

However, federated learning present a few challenges due to the key properties of federated optimization. We summarize these challenges by two key problems. Our first significant problem is communication overhead. Given the highly distributed nature of federated learning and the speed of end user device network connections, it is no surprise that communication overhead is a significant issue. More specifically, we surrender throughput and latency due to slow and expensive connections. We define latency as the time needed to transfer data over the network. Throughput can be defined as the amount of data that can be sent or received within a unit of time. The regular communication needed between server and participating devices may also result in network congestion resulting in added communication overhead. Thus, the need to communicate less between server and participating devices arises due to higher latency, reduced throughput, and an increase in network congestion. But, how do we determine the optimal communication strategy and what effect does this have on the overall learning process? One of the main conundrums in a federated learning setting is balancing computation and communication. More communication results in improved model accuracy but creates communication bottlenecks. Less communication has the opposite effect. In a federated learning setting, it is infeasible to communicate parameters every iteration of the learning process. We define these parameters in Section 2.1. There have been a few works that have explored periodic averaging to reduce communication and increase computation locally such as FederatedAveraging (McMahan et al., 2016; Stich, 2018). They even go as far as addressing the extreme case, one-shot parameter averaging. However, with less communication, there is slight degradation in model accuracy.

Our second problem is the problem that decentralized data brings to machine learning. Decentralized

data is great for user privacy but introduces the possibility of the data skewness problem which we cover in Section 2.4. We see that one of the state-of-the-art techniques, FederatedAveraging (McMahan et al., 2016) is not designed to address the problems related to data skewness. Evidently, these two conundrums are not trivial to solve.

If we break down balancing communication and computation, it comes down to investing communication efficiently. Averaging periodically still invests communication with clients independent of whether a model has converged to an optimum (Kamp et al., 2018). We define a client as any device or digital platform that participates in computation for a distributed machine learning task. Logically, we can invest more utility from clients that will contribute more towards a global model. (Kamp et al., 2018) deems this as **dynamic averaging**. At the heart of any machine learning task is the data. However, in federated learning, models are pushed to edge devices and run on decentralized data that is not well understood. There has been little exploration into understanding the underlying structural properties of the local datasets in a federated learning setting. There are popular methods such as singular value decomposition (SVD) to disentangle the factors of variance underlying the data and analyse certain properties of the data (Goodfellow et al., 2016). This poses the first question we want to answer in this study. Can the underlying structural properties of decentralized data inform a dynamic averaging framework that invests communication efficiently with negligible impact on model performance in a federated learning setting?

Alternatively, we can look to boosting algorithms such as AdaBoost (Friedman, 2000) as inspiration to invest communication efficiently. AdaBoost weighs observations, putting more weight on misclassified samples and less on those that have been classified correctly. Similarly, we can weigh communication for clients involved in a federated learning task. Clients that have misclassified more samples from their decentralized dataset communicate more and less when clients have less misclassified samples. This is desirable since focusing on misclassified samples in machine learning allows the learner to correct those mistakes in fewer iterations. In a federated learning context, this means fewer communication rounds and communication is invested efficiently. This brings us to our next question. Can we inform an optimal dynamic averaging framework based on the local loss of each client?

The main contributions of this research study are techniques that aim to invest communication efficiently in a federated learning setting. Two algorithms are presented in this study for decentralized learning. Firstly, we explore certain underlying structural properties of decentralized data to inform a

dynamic averaging framework to invest communication efficiently. We do this by not sharing data back to a centralized server. Secondly, we explore a dynamic averaging framework informed by the loss each client encounters similar to a dynamic averaging method presented by (Kamp et al., 2018). We discuss the differences of (Kamp et al., 2018) and the algorithms presented in this study in Section 2.4. There has been little work to invest communication efficiently in a federated learning setting. More importantly, there has been no approach that has explored the underlying structural properties of the local datasets to inform a dynamic averaging framework. This study aims to contribute to dynamic averaging methods by improving how communication is invested. We use different models ranging from logistic regression to multi-layer feedforward neural networks as a means of demonstrating the efficacy of this framework. More so, we provide a framework that optimizes how communication is invested in a federated learning setting. We explain the nature of these models in Chapter 3. It is also important to highlight that these methods can also extend to centralized learning environments since they often involve distributed networks. Distributed networks bring about similar bottlenecks to federated learning which may be solved by similar methods.

The techniques provided in this study do have their constraints. In federated learning, we may expect devices to have different hardware specifications and hence, different model update rates. These devices are considered to be heterogenous devices. The results presented in this study do not present experiments that have been tested with heterogenous devices. We limit this research study to homogeneous devices whereby update rates are similar due to the resource manager used in this study. This is discussed more in detail in Chapter 4.

The rest of the dissertation is structured as follows. We provide background into the main aspects of this dissertation in Chapter 2 providing related references. Thereafter, we introduce a detailed description of the dynamic averaging framework informed by local dataset properties as well as the assumptions we make in Chapter 3. The results, discussion and limitations of the framework follow in Chapter 4, and finally we conclude and give details about work to be considered in the future in Chapter 5.

Chapter 2

Background and related work

In recent years, there has been exceptional growth in commercial and academic machine learning datasets. As a result, researchers have explored large-scale machine learning to be able to cope with the abundance of data (Recht et al., 2011; Dean et al., 2012; Gopal & Yang, 2013; Abadi et al., 2016a). There are a number of properties that make up a large-scale machine learning problem. We know this traditionally involves centralized data, distributed optimization and inter-machine communication. However, due to user privacy concerns, the key properties differ slightly. The details are given in this chapter.

2.1 Baseline algorithms

The goal of a machine learning algorithm is for an algorithm to learn from data. There are three main subtopics of machine learning algorithms. Namely, supervised learning, unsupervised learning and reinforcement learning (Alpaydin, 2020). We focus on supervised learning in this study due to the wide coverage of academic research. Formally, we can define supervised learning as follows. Let's assume that we have a training set $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{n}$ where $\mathbf{x}^{(i)}s$ are the input variables also known as feature variables, $\mathbf{y}^{(i)}s$ are the target variables and $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ is the i^{th} training example from n samples. Essentially, the goal of a machine learning algorithm, given a space of input variables X, and space of target variables Y, is to learn some function $h : X \rightarrow Y$ so that $h(\mathbf{x})$ is a good predictor of $\mathbf{y}$ for a corresponding value of $\mathbf{x}$. There are two problems that supervised learning is able to solve:

- Classification problems if $\mathbf{y}$ is made up of discrete values; and
- Regression problems if $\mathbf{y}$ is made up of continuous values.

The type of problems we consider in this study are classification problems given the growth in academic research and availability of academic datasets for classification. Now, in order to estimate the parameters of $h(\mathbf{x})$, we first introduce what optimization is. Optimization can be seen as finding

one or more minimizer(s) of a loss function subject to constraints. A general optimization problem takes the form

$$\min_{\mathbf{w}\in\mathbb{R}^d} f(\mathbf{w}), \tag{2.1}$$

where $f(\mathbf{w}) = \ell(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \mathbf{w})$, that is, the loss of the prediction on example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ made with model parameters $\mathbf{w}$. We can consider an example in order to expand on the notion of optimization of a machine learning algorithm. Before we do that, we cover how we define $h(\mathbf{x})$ as a good predictor. Thereafter, we can explore the characteristics of machine learning mathematical optimization in a distributed environment in Section 2.2.

Overfitting and underfitting. We know that in a typical machine learning task, given some input variables $\mathbf{x}$, we need to learn a good predictor $h(\mathbf{x})$. A question that arises is how we define $h(\mathbf{x})$ as a good predictor. We introduce the concept of generalization. We define the ability of a model performing well on data that it has not seen as generalization. Generalization is a core challenge in machine learning (Goodfellow et al., 2016).

Typically, in machine learning, we split our given input variables into a training set and test set or validation set. The purpose of splitting the dataset is to be able to evaluate our model on unseen data. In practice when utilizing the predictor, we are likely to encounter unseen data. By being able to assess how well the model is expected to perform is thus key. Given our training set $(\mathbf{x}^{(train)}, \mathbf{y}^{(train)})$, we can compute a measure of error to get an indication of how well our model is performing. We introduce different error measures for which we define as loss functions later on. The error measure calculated using the training set is commonly referred to as the training loss. This process of reducing our error measure is the optimization problem we described above. However, what separates machine learning from optimization is that we also want to minimize the validation loss. Validation loss is defined as the expected error that the model is to incur on previously unseen data. The validation set $(\mathbf{x}^{(valid)}, \mathbf{y}^{(valid)})$ we describe is utilized to obtain the validation loss.

But how do we affect the validation loss if we are only observing the training set to obtain our mathematical parameters? There are a few assumptions that we can make which we borrow from statistics. We assume that the data isn't collected randomly and that we can make some assumptions about how the data is collected (Goodfellow et al., 2016). The process of collecting data generated

by a probability distribution is known as the data generating process (Goodfellow et al., 2016). We make two assumptions about this data. Firstly, we assume that observations in each dataset is **independent** from each other. Moreover, we assume that the training and validation set are **identically distributed**. Simply put, this means that the underlying distributions of the two completely separate datasets should be similar to one another. We refer to these assumptions as **i.i.d**. This is important to understand in order to understand the properties of federated learning explored in Section 2.4.

We cycle back to our question of how we define $h(\mathbf{x})$ as a good predictor. There are two factors that determine how a machine learning model has performed (Goodfellow et al., 2016):

1. Our training loss should be small; and
2. The difference between our validation loss and training loss should be small.

This brings us to the concepts of underfitting and overfitting. We define underfitting as the model being unable to learn the optimal parameters and is unable to achieve a small training error. Overfitting is defined as the model being unable to learn a set of parameters on the training set that translate to the validation set in a similar manner. In other words, the difference between training loss and validation loss is too large.

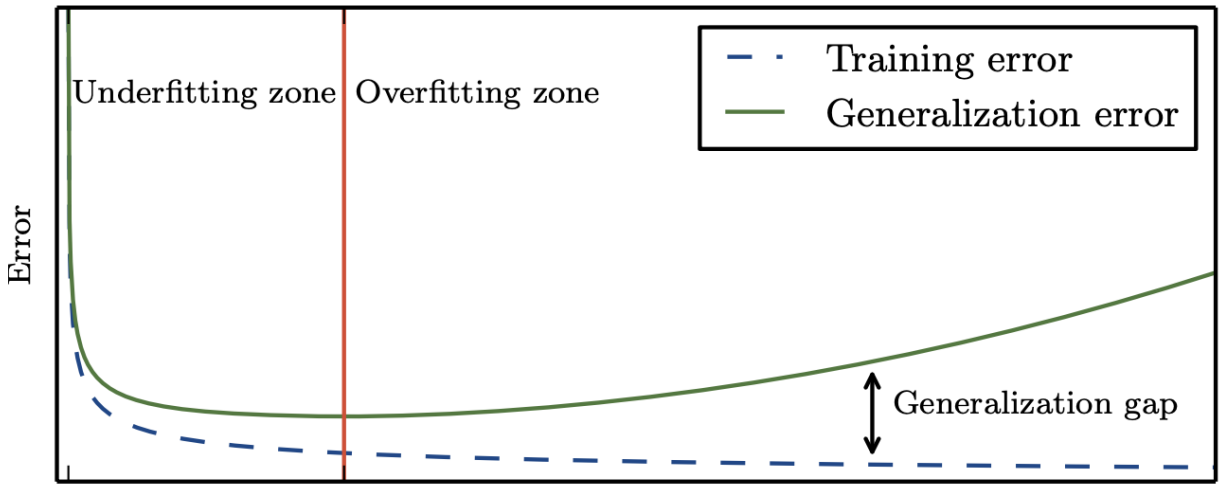

Figure 2.1: Visual of underfitting and overfitting (Goodfellow et al., 2016). We wish to achieve the smallest difference between training and validation loss whilst also obtaining the smallest training error. The red vertical indicates optimal zone we wish to obtain.

Each model we introduce in the rest of this section handles underfitting and overfitting in slightly

different ways. We briefly introduce these details in each of the sections to follow. We also keep in mind how this is an important thing to consider in a federated learning setting. In the next section and the sections thereafter, we introduce the different models we utilize for this study.

The different models that we cover in this study were chosen to provide good coverage of different model spaces. We cover more traditional models such as logistic regression and neural networks which have been growing in popularity in recent years. Also, both logistic regression and neural networks have been well researched in a distributed learning environment making them suitable choices for this study. We do not consider more complex model architectures to allow us to focus on the elements related to federated learning. That being said, it is possible to translate techniques discussed in this research study to complex models.

Logistic regression. Logistic regression is perhaps one of the simplest supervised machine learning algorithms. This allows for easy testing of different optimization methods in a large-scale setting. The tasks associated with logistic regression are classification problems. We can define the problems for logistic regression as follows. Given a set of training instance-label pairs $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{n}$, how do we obtain parameters of $h_{\mathbf{w}}(\mathbf{x})$, where

$$h_{\mathbf{w}}(\mathbf{x}) = g(\mathbf{w}^{\intercal}\mathbf{x}) = \frac{1}{1 + \exp^{-\mathbf{w}^{\intercal}\mathbf{x}}}. \tag{2.2}$$

A commonly used loss function for logistic regression is referred to log-loss or l1-loss, which is defined as follows:

$$\min_{\mathbf{w}} f(\mathbf{w}) = \frac{1}{n} \sum_{c=1}^{M=2} -\mathbf{y}_c \log(h_{\mathbf{w}}(\mathbf{x}_c)) - (1 - \mathbf{y}_c \log(1 - h_{\mathbf{w}}(\mathbf{x}_c))). \tag{2.3}$$

This loss function is commonly used for binary classification where we have $M = 2$ classes. If we consider multi-class problems then the loss function differs slightly. For M classes, then we utilize Eqn. (2.4) as our loss function.

$$\min_{\mathbf{w}} f(\mathbf{w}) = -\sum_{c=1}^{M} \mathbf{y}_c \log(h_{\mathbf{w}}(\mathbf{x}_c)). \tag{2.4}$$

Artificial neural networks (ANNs). Over the years, neural networks have progressed rapidly due to the computational power available in modern times. Much like other machine learning algorithms, the goal of a neural network is to estimate some function $h(\mathbf{x})$ such that $h(\mathbf{x})$ is a good predictor of $\mathbf{y}$ for a corresponding value of $\mathbf{x}$. We want to learn the best parameters $\mathbf{w}$ that result in the best function approximation. These functions are typically the loss functions that we have described above.

Neural networks are defined as networks since they are represented by a composition of different functions (Goodfellow et al., 2016). We can therefore represent any neural network architecture as a directed acyclic graph also commonly referred to as the computational graph. Let's say we have two functions $f_1(\mathbf{x})$ and $f_2(\mathbf{x})$ that are connected in sequence or a chain to produce $f(\mathbf{x})$. In other words, $f(x) = f_2(f_1(\mathbf{x}))$. We can represent this chain by Figure 2.2.

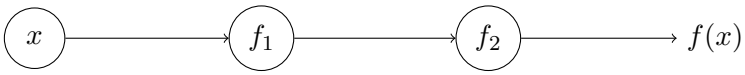

Figure 2.2: Directed acyclic graph for functions f_1 and f_2 given some input $\mathbf{x}$.

If we look at Figure 2.2 from a neural network point of view, we say that $\mathbf{x}$ is the **input layer**, f_1 is the first layer and f_2 is the second layer. In this case, f_2 is also the **output layer** since it produces the output $f(\mathbf{x})$. We can have multiple functions f_n that produce n such layers. This is also referred to as the depth and where "deep learning" gets it's name from. It is also common to refer to f_1 as the **hidden layer** since it does not show the desired output of the layer itself. Hidden layers can be seen as a layers that break the problem into it's constituent parts. There are many examples in image recognition that show the hidden layers detect edges in one layer and then other image features in the other layers.

There is also a mathematical function that is typically used in a neural network computational graph called the **activation function**. They act as a mathematical gate which decides whether a node in the graph should be activated or not based on their output. These are the fundamentals in setting up a neural network architecture. We must design our architecture including the number of hidden layers, how many units in each layer to use, and how these layers should be connected to one another. We must also decide on what activation functions to use in each layer. All these elements provide a more complicated optimization problem in order to learn the optimal parameters $\mathbf{w}$. Before we discuss how we learn those parameters in a complicated space, let us consider logistic regression as neural network to understand it's neural network architecture. Figure 2.3 shows how we can represent

logistic regression as a simple neural network.

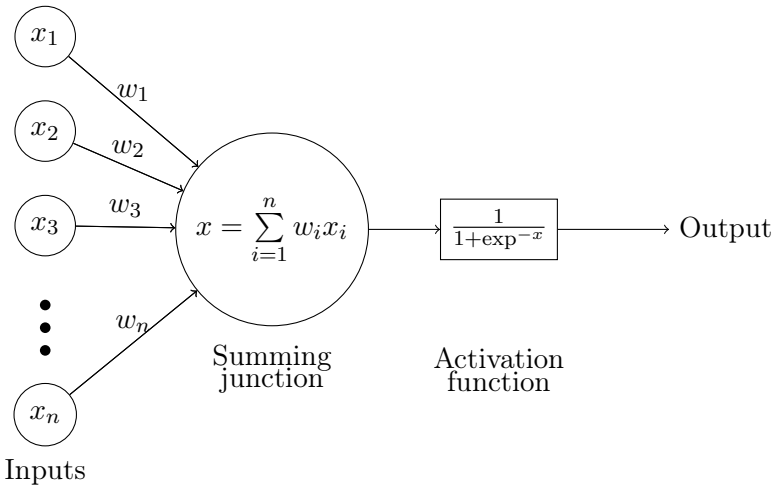

Figure 2.3: Logistic regression represented as a directed acyclic graph/neural network representation. The activation function for logistic regression is a sigmoid function. This is one of the many different activation functions.

We have an input layer which can be seen as our features $\mathbf{x}$. The second layer consists of summing junction and the sigmoid activation function. This is also known as the output layer in this case. These two functions are shown separately in Figure 2.3 for conciseness. A summing junction is simply the dot product of the inputs of the nodes and the weights of the edges. The sigmoid activation function squashes the output values of the summing junction to a range between 0 and 1. This provides us with expected outputs for each class label $\mathbf{y}$. We now have an indication how we perform a **forward pass** through a neural network architecture to obtain our network outputs. As discussed, we can increase the number of layers, units per layer and select different activation functions. Figure 2.4 shows a general architecture for a neural network.

In order to understand how we update our parameters through backpropagation, we first introduce the loss function one would typically use for such a problem and thereafter gradient based learning.

The type of problems we consider in this study are classification problems. Therefore, we only mention loss functions utilized for classification. We showed that Eqn. (2.3) can be utilized for logistic regression. This loss function is typically used for binary classification problems that extend wider than for just logistic regression. For multi-class classification problems we showed that we can use Eqn. (2.4) as our loss function. We can consider the same loss functions that are utilized for logistic regression i.e binary log-loss and multi-class log-loss, for neural networks. It is important to note

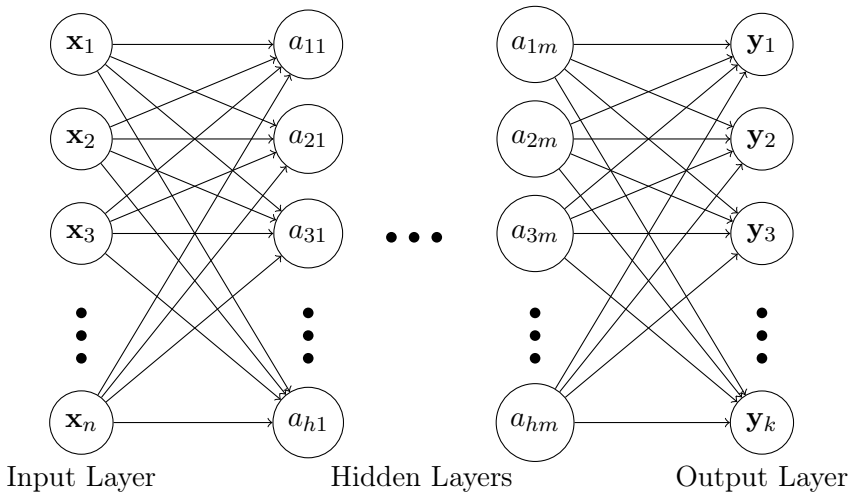

Figure 2.4: Illustration of general neural network architecture. The summing junction followed by an activation function is represented by the hidden layers a_{hm} where h is the number of hidden units and m is the number of hidden layers. The output layer is also a summing junction followed by an activation function represented by $\mathbf{y}_k$.

that even though we use the same loss function, the function approximation is different. In the next sections we discuss this.

Gradient based optimization. In order to minimize our loss functions defined above, we need to perform mathematical optimization (defined by Eqn. (2.1)). This brings us to gradient based optimization, more specifically SGD, which performs a stochastic approximation of gradient descent optimization. SGD is one of the first order methods for gradient based optimization that has typically been the conventional method and played a central role in large-scale machine learning (Bottou et al., 2016). However, SGD methods require carefully tuning the step-size and generally suffer from slow convergence (Gopal & Yang, 2013). For a long time, the natural first step was to improve on vanilla SGD (Ruder, 2016) optimization in order to achieve faster convergence.

There have been a number of alternative gradient methods to address the caveats of SGD. Noise reduction methods address the prevention of convergence to a solution for SGD when fixed stepsizes are used. They also address the slow sublinear rate of convergence when diminishing stepsizes are utilized. They contain 3 classes of methods: dynamic sampling methods, gradient aggregation and iterate averaging (Bottou et al., 2016). The primary tradeoff, in addition to greater computational cost, is the extra storage that we need to have in order to keep the extra state to compute the search

direction. Moreover, these methods are not widely used.

Other popular first order methods such as gradient descent (GD) methods with momentum, accelerated gradient methods and adaptive learning rate methods overcome the problems that SGD suffer from (Bottou et al., 2016). However, GD with momentum blindly follows the slope giving us highly unsatisfactory results (Ruder, 2016). Adagrad (Duchi et al., 2011), is an adaptive learning rate method, well suited for sparse data. However, it suffers from shrinking learning rates leading to the algorithm losing the ability to learn efficiently (Ruder, 2016).

On the other hand, second order methods such as Newton's method take into account curvature information. Thus, they provide exponential convergence near the optimal solution. However, calculating the inverse Hessian matrix is unmanageable in large-scale settings. Second order methods such as L-BFGS have high cost per iteration despite fast convergence (Lin et al., 2007). Quasi methods are second order methods that were introduced to overcome this problem by updating an approximation of the Hessian inverse (Bottou et al., 2016).

In addition, it is possible to leverage both first order and second order methods (Sohl-Dickstein et al., 2014). As mentioned before, SGD has its caveats and so do second order methods. By combining first and second order methods, we are able to leverage benefits from both methods if managed correctly. Sohl-Dickstein et al. (2014) do so by alternating between the quadratic approximation of the full objective, and evaluating a single sub-function and updating the quadratic approximation for that single sub-function for each optimization. Moreover, the memory and computational tractability are maintained.

There are more efficient first order methods that solve most of the caveats described above. We know that higher-order optimization methods such as previous works (Gopal & Yang, 2013; Lin et al., 2007) are ill-suited in high dimensional spaces. There have been some promising works to address these problems, such as Adam (Kingma & Ba, 2014). Kingma & Ba (2014) present Adam which focuses on the adjustment of the learning rate in a first order context much like the work of Hsia et al. (2017). Their method computes the individual adaptive learning rates for different parameters from estimates of first and second moments of the gradient. It combines the advantages of two distinguished optimization methods, RMSprop (Hinton, 2012) and Adagrad. RMSprop is an unpublished adaptive learning rate method introduced to resolve Adagrad's radically diminishing learning rates. Adam, being a first order method, does not suffer from memory requirements compared

to previously mentioned quasi-Newton methods (Gopal & Yang, 2013) based on mini-batches. This is often ill-suited to memory-constrained systems such as a GPU. More importantly, Adam is well suited for a large-scale setting and high dimensional parameter spaces.

There have been a number of alternative gradient methods to address the caveats of SGD (Lin et al., 2007; Duchi et al., 2011; Kingma & Ba, 2014). Despite these caveats, we rely on SGD for this study due to it's simplicity in the federated learning setting. However, it is important to note that the methods in this study can be extended to alternative gradient methods.

SGD. For optimization problems, we often consider the stationary points of a loss function. Gradient descent methods allow us to obtain these stationary points. SGD is an iterative method that has been widely used for a number of years to solve Eqns. (2.3) and (2.4). We briefly introduce SGD in the rest of this subsection.

At time step t, given the current model parameters $\mathbf{w}$, SGD obtains the updated parameters for Eqn. (2.2) by minimizing the loss functions which are shown in Eqn. (2.3) for binary classification and Eqn. (2.4) for multi-class classification. When used to minimize Eqns. (2.3) or (2.4), SGD performs the following update rule:

$$\mathbf{w} := \mathbf{w} - \alpha \nabla_{\mathbf{w}} f_t(\mathbf{w}), \tag{2.5}$$

where α is the learning rate or step size and $\nabla_{\mathbf{w}} f_t(\mathbf{w})$ is the vector of partial derivatives of f, with respect to $\mathbf{w}$ evaluated at time step t.

Adaptive Moment Estimation (Adam). SGD usually finds a local minimum, but does take longer than other gradient descent variants to converge. Moreover, SGD is reliant on robust intialization and has been known to get stuck in saddle points rather than local minima. In order to achieve faster convergence, we consider adaptive learning rate methods. Adam Kingma & Ba (2014) is one of the adaptive learning rate methods for gradient descent. Over the years, it has proven to be the go to optimization technique to achieve faster convergence. We look to briefly explore Adam.

Assume that the loss functions used in this section are the same loss functions described in Eqns. (2.3) and (2.4). We are interested in minimizing the expected values of these functions. Moreover, we can denote the gradient $d\mathbf{w} = \nabla_{\mathbf{w}} f_t(\mathbf{w})$ i.e. the vector of partial derivatives of f_t, with respect to $\mathbf{w}$

evaluated at time step t.

Adam benefits from both gradient descent with momentum and RMSprop. Adam stores an exponentially weighted (moving) average (EMA) of gradients $v_{d\mathbf{w}}$ in which the weighting for each older gradient decreases exponentially, similar to momentum. Bias correction aims to make EMA more accurate by bumping up the initial values of $v_{d\mathbf{w}}$ as shown in Eqn. (2.8). However, once time step t is sufficiently large the bias correction factor has no effect. Moreover, Adam stores an exponentially decaying average of squared gradients $v_{d\mathbf{w}}$ used to compute adaptive learning rates for each parameter. The moving averages themselves are estimates of the first moment (the mean) and the second raw moment (the uncentered variance) of the gradient.

We compute the decaying averages of past and past squared gradients $v_{d\mathbf{w}}$ and $S_{d\mathbf{w}}$ respectively, where the hyper-parameters β_1, $\beta_2 \in [0, 1)$ control the exponential decay rates of these moving averages. The details are shown in Eqns. (2.6) and (2.7):

$$v_{d\mathbf{w}} = \beta_1 v_{d\mathbf{w}} + (1 - \beta_1) d\mathbf{w}, \tag{2.6}$$

$$S_{d\mathbf{w}} = \beta_2 S_{d\mathbf{w}} + (1 - \beta_2) d\mathbf{w}^2, \tag{2.7}$$

where $v_{d\mathbf{w}}$ and $S_{d\mathbf{w}}$ are first and second moments of the gradients, respectively. Since both $v_{d\mathbf{w}}$ and $S_{d\mathbf{w}}$ are initialized as vectors of 0s, the authors of Adam implement bias correction as per Eqns. (2.8) and (2.9):

$$v_{d\mathbf{w}}^{corrected} = \frac{v_{d\mathbf{w}}}{1 - \beta_1^t}, \tag{2.8}$$

$$S_{d\mathbf{w}}^{corrected} = \frac{S_{d\mathbf{w}}}{1 - \beta_2^t}. \tag{2.9}$$

Finally, these formulations are used to perform a parameter update as follows:

$$\mathbf{w} := \mathbf{w} - \alpha \frac{v_{d\mathbf{w}}^{corrected}}{\sqrt{S_{d\mathbf{w}}^{corrected} + \epsilon}}, \tag{2.10}$$

where $\epsilon > 0$.

Backpropagation. We introduced neural networks by showing how these functions we estimate can be represented by a chain of functions. We also introduced gradient based methods to show how we can solve for parameters $\mathbf{w}$. Backpropagation is the method we use to compute the gradients. We use a gradient based learning algorithm such as SGD to perform learning using the calculated gradient. Intuitively, we can describe backprogation as sending the information that we obtained from our loss function back through the network in order to calculate the gradient. If we look at it from a mathematical perspective, we see that this becomes a calculus problem. In gradient based learning, we calculate the gradients of the loss functions with respect to the parameters $\nabla_{\mathbf{w}} f(\mathbf{w})$. Our computational graph contains a composition of functions. If we recall our differentiation rules from calculus, we see that this can be computed using the chain rule. The chain rule says:

$$\frac{d}{dx}[f(g(x))] = f'(g(x))g'(x). \tag{2.11}$$

The chain rule tells us how to differentiate composite functions. From a neural network point of view, we wish to optimize $\mathbf{w}$, therefore we use the chain rule and differentiate with respect to $\mathbf{w}$.

We ignore other details regarding how we can improve our neural networks since this is not the primary focus of this study.

Convolutional neural network (CNNs/ConvNets). Regular neural networks do not scale well when we consider images as input. Let us consider an image of size 28 pixels by 28 pixels. Also, let's say that this image has 3 colour channels. This will result in $28 \times 28 \times 3 = 2,352$ weights. This seems manageable. However, if we scale up the image size, say 200 pixels by 200 pixels with 3 colour channels, we obtain $200 \times 200 \times 3 = 120,000$ weights. When we add more hidden units, and hidden layers this will quickly add up and become unmanageable. Convolutional neural networks (Le Cun et al., 1989) were introduced to solve this problem. They are similar to neural networks that we introduced in the previous section. They are also made up of weights and biases and use similar operations such as activation functions and dot products. However, the architecture differs slightly and introduces another operation. CNNs were specially constructed for data that appears to be of a grid-like topology. Exampes include time series data, and image data. The name **convolutional** neural network suggests that the network includes a convolutional operation.

A convolution is an operation on two functions that produces a function that describes how the shape

of one function is described by the other. Mathematically we can define the convolution operation by Eqn. (2.12)

$$(f * g)(t) = \int f(\tau)g(t-\tau)d\tau, \tag{2.12}$$

where $(f * g)(t)$ is the convolution result at some moment t and τ is a dummy variable (Goodfellow et al., 2016). In the case of machine learning, it is some variable in the feature set $\mathbf{x}$. The function g is considered as a weighting function that gives weight to more recent observations. Applying this weighted function at every moment provides a smoothed estimate of our function f. It is important to consider that g needs to be a valid probability density function, else the output we obtain will not be the weighted average (Goodfellow et al., 2016).

If we translate Eqn. 2.12 to a machine learning context, function f translates to inputs or feature dataset. Function g is considered to be the **kernel**. The primary function for the convolution operation is to extract features from data that has a grid-like topology. Goodfellow et al. (2016) provides a graphic that explains this well which is shown by Figure 2.5.

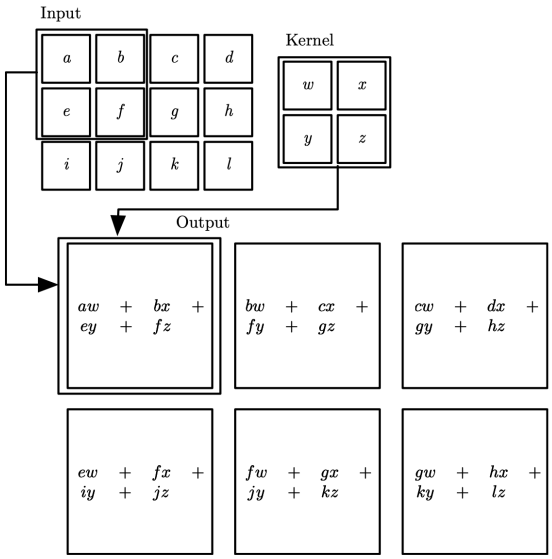

Figure 2.5: Intuitive explanation of the convolutional operation provided by Goodfellow et al. (2016). The boxes and arrows show how the top-left elements of the output are calculated using the corresponding top-left elements of the inputs and the kernel.

The size of the convolved feature shown by Eqn. (2.12) is controlled by three parameters we list below. We skip over detailed explanations of the three parameters shown below.

- Depth: The number of kernels we use for the convolution operation,
- Stride: The number of pixels we slide our kernel over the input matrix,
- Zero-padding: It may be convenient to pad the inputs with zeros along the borders making it possible to apply the kernel along the boundary.

A typical convolutional neural network consists of three stages. The first is the aforementioned convolution operation. This gives us a set of linear functions. Stage two involves running each linear function through some non-linear activation function such as the sigmoid function in Figure 2.3. In stage three we introduce a **pooling** layer. A pooling layer replaces the output at a specific position with a aggregation statistic of it's nearby outputs (Goodfellow et al., 2016). For example, **max pooling** replaces the output at a specific position with the maximum output within a rectangular perimeter. Pooling improves computational efficiency by selecting the most important information.

Once we setup the architecture that follow the three stages, we then add a fully connected layer and an output layer. This is equivalent to our hidden layers we described using Figure 2.4. All these stages along with a fully connected layer gives us the complete architecture of a simple convolutional neural network.

We've covered the baseline algorithms for this study of federated learning. In the next section we cover large-scale machine learning in a distributed environment. This is an important aspect in a federated learning setting.

2.2 Distributed setting

Large-scale machine learning in a distributed environment has a few different characteristics as opposed to large-scale machine learning algorithms on a single machine. Firstly, it is necessary to distribute the optimization of a machine learning algorithm. Secondly, in order to orchestrate distributed optimization we require inter-machine communication. We discuss this in detail in a data centre context in the following sub-sections.

Distributed system. Whilst optimization is a key aspect to distributed large-scale machine learning, the characterization of scientific applications and architectures is a vital process in large-scale settings. Due to the growth of data in the natural world, many problems are too large and/or complex to solve on a single machine. Given the limited memory available on a single machine, it becomes impractical

or impossible to solve large-scale problems (Blaise Barney, 2008). However, implementing an efficient distributed algorithm is not trivial. The issues that arise such as the volume of data communication and the computation workload demand meticulous system design (Xing et al., 2016).

Parallel programming model. We briefly explore a few key aspects of a parallel programming model in order to handle the deluge of data (Blaise Barney, 2008):

- Data parallel,
- Single program multiple data (SPMD),
- Model parallel.

Data parallel. A data-parallel approach refers to the same task being applied on a different partition of the data (Blaise Barney, 2008). In this context, the same task refers to the same machine learning model. We can achieve greater speedup and it is ideal for array and matrix computations. This is because the data is partitioned amongst the nodes and computations are performed on smaller partitions on each node in parallel. This approach is suitable to enable optimum load balancing of work and can be built upon to make the system more flexible.

Model parallel. A model parallel approach refers to the same data being used for different portions of the model computation split among computational clients simultaneously (Abadi et al., 2016a). This programming model is also available in Tensorflow. For simpler algorithms such as linear and logistic regression, it doesn't seem necessary to use this paradigm. This approach becomes useful when extending the system to deep learning.

Single program multiple data (SPMD). SPMD can build upon any combination of the previously mentioned programming models. It has the logic programmed into them to execute only those portions of the program that are designed to execute (Blaise Barney, 2008). This type of programming model is probably one of those most common models used (Blaise Barney, 2008). One of the architectures that follow this paradigm is the server-client architecture. The server usually executes the non-parallelizable code and the clients execute the portions of the model that is decomposable.

Designing parallel programs. Using one of the aforementioned parallel programming models requires much thought to ensure the full benefits of each programming model. We can parallelize a serial problem by breaking it down into four key steps (Solihin, 2015):

1. Decomposition - either the data and/or tasks are broken down into small tasks.
2. Assignment - each process is assigned a task.
3. Orchestration - refers to how data is accessed, communicated and synchronized.
4. Mapping - processes mapped to processors.

Decomposition and assignment. In order to achieve optimal results in our distributed machine learning algorithm, the allocation of data and computation needs to be carefully mapped to the available clients. Tensorflow (Abadi et al., 2016a) executes a placement algorithm. which simulates the execution of the system with a small number of data points to determine the optimal configuration. We see in Section 2.3 that this is by default in a federated learning setting and is not something controlled by a coordinator.

Orchestration. To control the volume of data communication, there are a number of factors to consider when designing a programs inter-task communications. Communication overhead, latency vs bandwidth, synchronous vs asynchronous, scope of communication, etc, form part of this partial list (Blaise Barney, 2008). As Vetter & Mueller (2003) and Hoefler & Schneider (2012) discuss, majority of data transfers occur in point-to-point communications. On the other hand, collective communications have small payloads but most time is spent in these operations (Klenk & Fröning, 2017). Neighbourhood collectives are one of the methods that can be used to address this issue. However, communication topology, sizes and buffer access constrain the optimization space.

In a distributed environment, clients must regularly read and update the global parameters. This can create huge communication overhead. Also, since optimization is inherently sequential, all machines would need to wait for synchronization. Therefore, asynchronous updates have also become key to efficiency, but need to ensure there is proven convergence. As a result, parameter server frameworks have become quite popular in distributed machine learning systems (Li et al., 2014a,b). In this case, Li et al. (2014a) present an asynchronous block proximal gradient method available in the parameter server framework. The relaxations available to this framework were controllable asynchrony via task

dependencies and user-definable filters. They are able to reduce data communication volumes. They experimented on several challenging tasks on real datasets up to 0.6PB size with hundreds billions of samples and features. However, the system has a high level of complexity. They addressed this in their later works (Li et al., 2014b). Globally shared parameters are used as local sparse vectors or matrices to perform linear algebra operations with local training data.

There exists alternative communication efficiency strategies such as quantization. Sufficient factor broadcasting (SFB) (Xie et al., 2015) improves communication efficiency by broadcasting the "sufficient factors" (SFs) amongst clients and reconstructs the update matrices locally at each client. The parameter update computed on each data sample is a rank-1 matrix. i.e. the outer product of two sufficient factors (Xie et al., 2015). As a result, communication costs are linear in the parameter matrix dimensions, rather than quadratic. There have been a few alternative quantization methods that have aimed to reduce communication costs. Alistarh et al. (2016) present Quantized SGD (QSGD), which quantizes gradients by randomized rounding to a discrete set of values and efficiently generate efficient encodings. Compared to 1BitSGD (Seide et al., 2014; Strom, 2015), QSGD can achieve asymptotically higher compression, provably converges under standard assumptions, and shows superior practical performance in some cases. This study has not considered quantization in detail and is something that we wish to consider as future work.

Mapping. In a distributed environment, we assume that determining the optimal configuration by simulating activity in the system will allow us to map processes optimally. In fault tolerant systems, it would make sense to save the state of the model and restart the execution Abadi et al. (2016a). In federated learning, computation is cheap since the number of samples at a client is small. Therefore, it becomes more about client selection based on the properties discussed in Section 2.4.

Distributed gradient optimization. Optimization methods, both first and second order, are inherently sequential. Consequently, there has been significant attention to parallelize and distribute optimization in a data center context (Recht et al., 2011; Dean et al., 2012; Abadi et al., 2016a).

We have established that in parallel applications such as distributed optimization we need to consider communication costs as well as computation costs. We discussed earlier that asynchronous updates contribute to overall communication efficiency. However, applying this approach to SGD, runs the risk of divergence since there exists model replicas which send updates to the parameter server but

do not communicate with each other (Dean et al., 2012). Recht et al. (2011) provides an alternative asynchronous framework which presents a similar problem and is only optimal if the input data is sparse (Ruder, 2016). In order to overcome the risk of divergence Zhang et al. (2014) propose a deep learning with elastic averaging SGD algorithm. They reduce the amount of communication between server and clients, allowing the clients to fluctuate further from the global parameters. Their update rule for the global parameters use a moving average where the average is taken over both space and time. Thus, the model replicas at each client have a lower risk of divergence.

There also exists optimization algorithms that have a separable objective which means it is highly parallelizable. The alternating direction method of multipliers (ADMM) is a popular method that fits this description for distributed convex optimization (Chen et al., 2014). However, this means ADMM is reliant on the distributed configuration.

Second order methods have also been popular in a distributed large-scale setting. We have seen that generally second order methods suffer from high computation costs per iteration. It has also been proven that second order methods such as naive L-BFGS are not effective under large scale settings (Gopal & Yang, 2013). A number of methods have addressed this problem such as Chen et al. (2014). They propose an effective approach to scale up and parallelize the naive implementation of L-BFGS. The Vector-free L-BFGS system using MapReduce avoids the expensive dot product operations in the two loop recursion. However, there are more efficient frameworks than MapReduce. Zhuang et al. (2015) propose a distributed Newton method of training large-scale logistic regression. They conduct parallel matrix-vector products and even outperform ADMM. There has been little attention to the adjustment of the step-size for second order methods. Hsia et al. (2017) address the inappropriate step-size adjustment of the work (Lin et al., 2007) which approximates the gradient directions. However, Adam addresses this and is more efficient since it is a first order method.

We reiterate that combining first and second order methods suggests we could benefit from the positive characteristics of both. Much like Sohl-Dickstein et al. (2014), this methodology is scalable and efficient in a distributed environment i.e. VW (Gopal & Yang, 2013). VW combines SGD and quasi-newton methods. They apply a stochastic gradient method in the beginning, then switch to parallel L-BFGS (Zhuang et al., 2015) for a faster final convergence. However, Zhuang et al. (2015) prove that their method shows more practicality in the real-world in a distributed environment than VW.

It is also important to consider these methods in a fault tolerant environment since this is key in

distributed environments. Fault tolerance is the capability to operate and to recover loss after a failure occurs. Many of the works have not explored these optimization methods in a fault tolerant system. Lin et al. (2014) address this by exploring a Spark implementation of distributed Newton methods for solving logistic regression and linear SVM (Zhuang et al., 2015). They achieve good results relative to existing Spark MLLib implementations, but do not achieve speedup over their original MPI implementation.

Distributed methods without gradients. While gradient methods have been widely used for optimization, Taylor et al. (2016) propose a Scalable ADMM approach without gradients for training neural networks. They aim to address the problems of gradient methods when scaling up to a large number of cores. Moreover, the convergence of gradient methods suffer from saturation effects, poor conditioning and saddle points. They propose an unorthodox method that uses alternating direction methods and Bregman iteration to successfully train neural networks without gradient descent steps. They reduce the problem of training network parameters to a series of minimization sub-problems using ADMM which are solved globally in a closed form. They also overcome the limitations that make gradient methods slow on highly non-convex problems. Their method is promising, as they achieve linear speedups even when scaling up to thousand of cores which systems like Downpour SGD struggles to do so (Dean et al., 2012).

We mention once again, that despite it's caveats, we consider SGD in a distributed setting, deemed **parallel-SGD**, due to it's simplicity. The elements that we have explored thus far, namely, optimization and inter-machine communication in a distributed environment provide a good basis for Section 2.3. So far, we have considered distributed machine learning in a data center context where we have centralized data. On decentralized data, there are a few key elements that differ.

2.3 Federated learning

The type of problems that are suitable for federated learning are ones that require privacy at the heart of training models. Let us consider a few scenarios we would encounter in a federated learning setting for conciseness before explaining the details of federated learning. In our first scenario, we may have different militaries that have collected surveillance footage. Now, each military may have their own model to detect enemy operations, but we may obtain a better model if we combine these models. We know from ensemble methods that weak learners that explore vastly differently spaces that are

combined perform very well. Let us consider another scenario in the health industry. Health data is considered to be sensitive data and cannot be shared due to regulatory requirements. But once again, we may want to mine this data to extract valuable insights and combine them. The same goes for the financial industry. We may want to mine the overall data for risk analysis across banks. Once again we have regulatory requirements and great risks of transferring large datasets. We now understand the role that federated learning plays in society. Let us now explore the details of federated learning.

Typically, this method involves training a shared model under the coordination of a centralized server from a federation of participating devices (referred to as clients) on decentralized data (McMahan et al., 2016). Each client downloads the shared model, learns a new local update via SGD (Bottou et al., 2016) and shares the update with the server. The server aggregates these multiple updates from the clients to improve the shared model. This update is then shared with each client and repeats for a chosen E number of epochs. In comparison to training datasets, these updates do not need to be stored after usage since they generally contain less information than raw training datasets.

It is important to note that there are multiple ways of aggregating these updates in a federated learning similarly to machine learning on centralized data. We need to be cognisant of the properties of optimization in a federated learning setting to discuss the different ways of aggregating these updates. We explore these properties in Section 2.4. Thereafter, we look to see how we consider these properties in aggregating updates.

We assume a shared model since this simplifies the task of aggregating the multiple updates even though it is possible to convert some model architectures into another (Verma et al., 2019). This means, each client will learn the same type of model e.g. all clients learn a logistic regression model or all clients learn the same neural network architecture.

The resulting federated learning architecture is a typical server-client architecture which is explained in Section 2.2. Each client is responsible for doing some work. Then a client shares it's results with the server. The server is responsible for collating these results.

We can categorize federated learning into three categories; Horizontal federated learning, vertical federated learning and federated transfer learning (Yang et al., 2019). Horizontal federated learning is introduced in cases where datasets share the same feature space but have different samples (Yang et al., 2019). For example, let us consider two different companies that have different user groups from their respective regions but share similar business. There would be little overlap in the samples but

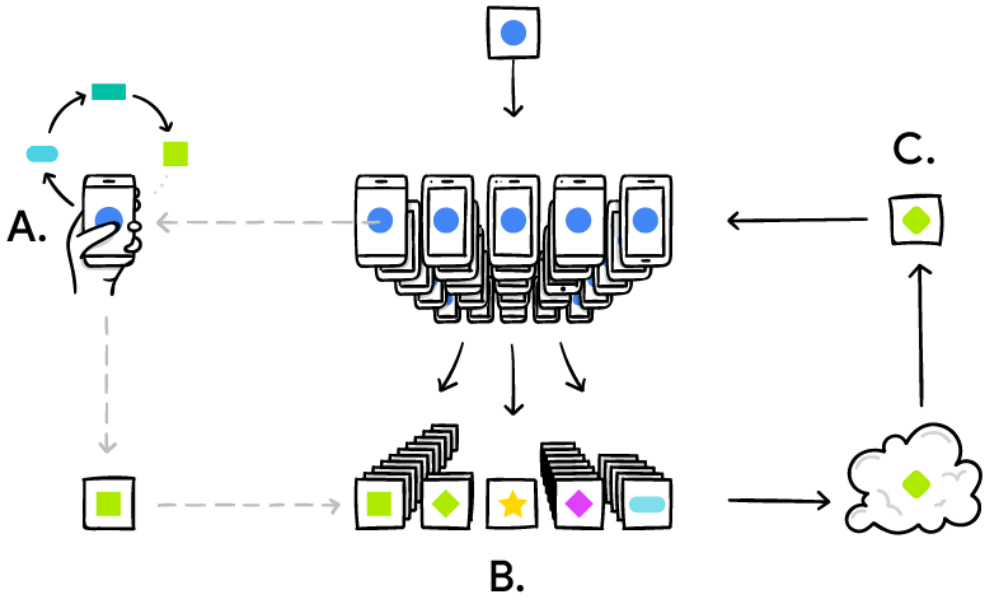

Figure 2.6: The federated learning process (Brendan McMahan and Daniel Ramage, 2017). Clients update their local models based on their usage (A). Updates are aggregated (B) and this change (C) is applied to the global shared model.

would have a similar feature since their businesses are similar. On the other hand, vertical federated learning is when datasets share the same sample ID space but have different feature spaces (Yang et al., 2019). A good example of this is when two different companies in different industries exist in the same city. They have a large intersection of their user base, but collect different information and hence a different feature space. Federated transfer learning refers to scenarios when two datasets not only differ in samples but also have different feature spaces (Yang et al., 2019). An example of this would be when there are two different companies in different industries that exist in different cities. Due to geographical restrictions there may exist only a small intersection resulting in different samples but also different feature spaces. In the upcoming sections, we explore the fundamental properties of federated learning that give an indication how the different categories of federated learning are solved.

2.3.1 Privacy

We previously mentioned that federated learning was introduced to solve problems where privacy is a core aspect. Let us explore this in more detail. Firstly, we briefly explain decentralized data. We explore the challenges that this brings to machine learning in Section 2.4. We also need to understand

what needs to be protected and as a result, briefly introduce differential privacy.

Decentralized data. Decentralized platforms have existed for a while given the rise of technologies such as decentralized source control and blockchain. This technology has also found its way into the machine learning ecosystem. We know that the abundance of user data available in current times has meant machine learning can flourish. Companies are able to store terabytes of data in their data centers and provide personalized experiences. However, with the absence of data protection laws in the early days, companies exploited this data to manipulate users in different ways. Data that was collected for a single use was used for other use cases without the user being aware. With the rapid development of mobile technology, mobile devices have significant computing power. This meant that the development of machine learning models could be shifted to the end user's device and data would not need to be shared with a centralized server. This data that sits on the end user's device is what we deem as **decentralized data**. This is since we do not have a single entity (some company A) that owns the data but rather the user owns the data. But this poses challenges for machine learning models which we explore in 2.4.

What needs to be protected. Vepakomma et al. (2018) lists the following aspects of datasets that protection mechanisms should protect:

1. Input variables;
2. Target variables; and
3. Identifiable information such as which party contributed to a specific record.

We defined the input variables and target variables in Section 2.1. The first two items in this list refer to the raw data itself. The third item in the list refers to identifiable information that may be derived from the input and target variables. This briefly introduces the concept of differential privacy.

Differential privacy. Machine learning techniques may consist of methods to protect their training data. Regularization techniques which assist in avoid overfitting may protect details of observations. However, in some cases, an adversary may be able to extract parts of the training data from the shared parameters (Abadi et al., 2016b; Kairouz et al., 2019). To our knowledge there are no current studies that have explicitly addressed the restrictions on sharing structural properties of the client-side

data which is core to Algorithm 3. However, there have been studies that have demonstrated model inversion on facial recognition systems (Fredrikson et al., 2015). This may require techniques such as differential privacy. We briefly introduce differential privacy to get an overall picture of the current state of federated learning. However, we do not explore differential privacy in detail in this study.

Differential privacy is a privacy preserving technique which ensures that data subjects can not be disclosed in a shared data set. It achieves privacy preservation by introducing noise to the data before it is shared. This reduces the ability to identify information as to which party contributed to a specific record.

There are a number of different ways to achieve differential privacy in a federated learning environment. It is possible to use random sub-sampling and distorting as a differential privacy technique (Geyer et al., 2017). Geyer et al. (2017) defines random sub-sampling as selecting a random subset of clients in each epoch. Distorting is a Gaussian mechanism used to distort the sum of all updates. By combining these two steps, we reduce the ability of identifying information. There are also mechanisms in SGD that we utilize to assist with model training such as clipping gradients. This can be used in conjunction with Gaussian mechanisms (Abadi et al., 2016b) for differential privacy. Principal component analysis is a dimension reduction technique which is considered to be hard to interpret since it transforms datasets to components in an Eigen space. This is suitable for differential privacy (Abadi et al., 2016b). There are additional considerations when including differential privacy in a federated learning environment. Kairouz et al. (2019) details some of the limitations of existing differential privacy methods. In particular, the challenges in the absence of a fully trusted server. We plan on exploring these challenges in future works.

It is also important to consider the computational costs of differential privacy. To get an indication of such costs, we take our distortion example described above. Generating random noise and adding it to real numbers i.e. our mathematical parameters, can be atomic. To generate a cumulative distribution function takes constant time $\mathcal{O}(1)$. Since we are looking to add random noise to our mathematical parameters, performing such a computation for n parameters takes $\mathcal{O}(n)$. However, adding noise may result in the gradient descent trajectory to be derailed slightly causing slower convergence. Thus, to get a clear indication of how privacy scales requires a far more detailed review of differential privacy works. We look to cover differential privacy in a federated learning environment in more detail in future works.

Robustness to Attacks and Failures In federated learning, the distributed nature and data constraints can open up new opportunities for attacks and make detecting and correcting failures a challenging task (Kairouz et al., 2019). In a distributed datacenter and centralized learning settings, it has been shown that model update poisoning, data poisoning and evasion attacks exist (Kairouz et al., 2019). In federated learning, this is no different (Kairouz et al., 2019). Kairouz et al. (2019) show that there have been techniques to address model update poisoning such as Byzantine-resilient defenses (Kairouz et al., 2019). Similarly, there are techniques to address data poisoning, namely, data sanitization and network pruning (Kairouz et al., 2019). However, there have been works that have shown that these techniques have vulnerabilities (Kairouz et al., 2019). This particular topic of federated learning is also something we wish to address in future work. In a federated learning environment, we know that failures are expected given the highly distributed nature. We can detect non-responsive clients and omit their updates from a communication round update for efficiently sake (Kairouz et al., 2019). However, if considering other aggregation techniques such as secure aggregation (SecAgg) (Bonawitz et al., 2017), this becomes difficult since SecAgg only allows the server to see an aggregate of the client updates, not any individual client updates. This highlights another key aspect in federated learning, namely the tension between privacy and robustness. We also look to explore this topic in future works.

2.4 Federated optimization

There are some key differences to data center optimization in a distributed setting highlighted in McMahan et al. (2016). We list the key properties for this study for conciseness in this section and thereafter address some other challenges that we encounter in a federated learning setting.

1. **Non identical and independent data (non-IID)** The dataset we have locally is not representative of the entire dataset.

2. **Unbalanced data** Certain clients/digital platforms may have produced more data, leading to varying amounts of data.

3. **Slow and expensive connections** Given the distributed nature of clients and speed of connections, communication overhead is a significant issue.

We cover both Non-IID and unbalanced data simultaneously due to the relationship of these properties.

Non-IID and unbalanced data. FederatedAveraging McMahan et al. (2016) has emerged as one of the leading methods for training non-convex models in the federated learning setting. Their work presented an approach that addresses the key elements of federated optimization listed in Chapter 1. However, Sahu et al. (2018) highlights that FederatedAveraging was not designed to tackle the statistical heterogeneity inherent in federated settings; namely, that data may be non-identically distributed across clients, with the number of data points per client varying significantly. It has also been shown to diverge empirically in this setting (McMahan et al., 2016, Sec 3).

It is possible to reduce divergence of local models by ensuring feature representation in the global (aggregate) model can be derived from locally learned representations (Mostafa, 2019). Reconstructing the feature representations locally means each client does not run the risk of learning representations that are too specific. In addition, it is possible to use adaptive on-line tuning of hyperparameters by adding a regularization term that penalizes divergent representation across clients (Mostafa, 2019). This is done through online reinforcement learning. In each round, the learners perform an action of selecting hyperparameters and at the end of each round get a reward which is the relative reduction in training loss. The hyperparameter selection is optimized through maximizing these rewards.

Moreover, a client can solve a surrogate function that effectively limits the impact of the local updates (Sahu et al., 2018). This minimizes the divergence caused by learning local representations that are too specific since we are keeping local updates close to the initial model. This technique is related to elastic averaging with SGD (Zhang et al., 2014) which we have mentioned previously under a data center setting.

If we consider function estimation when the data is statistically heterogeneous, we may find some samples in the data that refer to different ranges in the value of the function. Using FederatedAveraging for the model averaging process is likely to lead to an incorrect result (Verma et al., 2019). In order to overcome this issue, it is necessary to be aware of the region of applicability of the estimate. In other words, we average only those parts of the system that are in a similar region. This approach is called bounds-aware fusion (Verma et al., 2019). However, the bounds are not well understood for raw data such as images/audio where a neural network due to the distribution of feature space for such data.

Similarly, there are bounds-expanding data exchange methods to handle the heterogeneity of the data, more specifically the non-IID property. It is shown that even exchanging a limited number of data points increases accuracy. In the real world, this poses privacy risks. To overcome such

risks, Zhao et al. (2018) propose a strategy that creates a small subset of data containing a uniform distribution that is globally shared from cloud to all participating devices. The globally shared data is independent of the client's data so it is not privacy sensitive. Experiments using this strategy result in approximately 30% increase in accuracy on the CIFAR-10 dataset (Krizhevsky, 2009) with only 5% globally shared data. In addition, it is possible to consider differential privacy in a federated learning setting (Vepakomma et al., 2018) which may further assist in overcoming issues that come with bounds-aware data exchange. The use of differential privacy may allow for sending those data points that happen to be under represented at other nodes in a federated network.

So far, we have assumed that all clients have no missing features, missing classes or missing values, but realistically this may be the case. Verma et al. (2019) present approaches to reconstruct features/classes/values using popular techniques such as oversampling (Synthetic Minority Oversampling) and generative adversarial networks (GANs). In a federated learning environment, this poses a few challenges but can be explored to handle the non-IID problem.

Slow and expensive connections. The clients that exist in a federated learning setting usually are on slow and expensive connections. As a result, there are a number of algorithms that have addressed communication efficiency to avoid communication every round. There have been a few works McMahan et al. (2016); Stich (2018); Kamp et al. (2018) that have explored periodic averaging. Instead of communicating parameters every round as in data center optimization, communication only occurs every few rounds. We know that mini-batch SGD is one of the state of the art large scale distributed training techniques. In a distributed setting, this suffers from frequent communication and bandwidth limits. We can limit communication every round and only communicate every few rounds (Stich, 2018). It is possible to limit communication by up to a factor of $T^{1/2}$, where T denotes the total number of steps. This still shows similar convergence rates to mini-batch SGD (Stich, 2018).

However, this means that this technique still invests in communication regardless of whether the local models have converged to an optimum. Briggs et al. (2020) propose a method that allows model training to converge in fewer number of communication rounds (significantly so under non-IID settings). A hierarchical clustering step is introduced to separate clients into clusters by similarity of their local model updates to the global model. Once separated, these clusters are trained independently in parallel on specialized models. This results in improved performance over the original FederatedAveraging (Brendan McMahan and Daniel Ramage, 2017) protocol. Moreover, recommendations for the clustering

hyperparameters are provided. However, it is unknown whether these hyperparameters provide different results for different model architectures as their experiments are performed only with convolutional neural networks.

On the other hand, the authors of Kamp et al. (2014) address investing communication efficiently by only communicating from clients that violate some divergence threshold Δ. Learners that do not violate this divergence operator do not communicate until they meet this criteria. This is first explored in a convex setting in Kamp et al. (2014). The authors later explored this algorithm in a non-convex setting in Kamp et al. (2018). As mentioned, their dynamic averaging protocol includes communicating model updates only when a local model's divergence exceeds a divergence threshold Δ. Each local learner keeps track how far it deviates from a reference model r that is common amongst all learners. This change in the difference between updated model f and the reference model r is used to check if a client's local learner has violated the divergence threshold. The divergence threshold Δ is manually selected or is an additional hyperparameter to fine-tune. The first choice for a reference model is the model updates from the previous synchronization step. Moreover, this technique also addresses the non-IID and unbalanced properties through weighted model averaging, where the weight is the number of samples observed in each round. We provide this algorithm shown by Algorithm 1 since we use this algorithm as one of our benchmarks. We present a similar but alternative dynamic averaging method in Algorithm 4 presented in Chapter 3. The main difference will be highlighted in the corresponding chapter.

To reduce communication costs, one can also utilize subsampling. This involves randomly selecting clients to communicate their parameters. The server thus averages these subsampled updates (Konečný et al., 2016). We can also look to quantization methods in a federated learning setting. Recall that quantization is the process of mapping a large set of values to a smaller set of values. In this context, it is encoding and compressing the mathematical parameters when sending them to and from the clients and the server. Much of the quantization studies that have been applied in a data center setting can be translated to a federated learning setting. Sketching (Woodruff, 2014) is a form of quantization that can be utilized. Sketching is a technique whereby given a matrix, one first compresses it to a smaller matrix by multiplying it by random smaller matrix with certain properties (Woodruff, 2014). Another way of compressing the updates is by quantizing the parameters. We have shown in Section 2.2 that we can quantize parameters using more than 1-bit (Seide et al., 2014) using QSGD. QSGD provides a simple way of balancing between accuracy and communication costs. This also applies in a

Algorithm 1: Dynamic averaging protocol for unbalanced data

Input: divergence threshold Δ, batch size b, number of learners m

Initialization: local models $f_1^1, ..., f_1^m \leftarrow$ one random f

reference vector $r \leftarrow f$

violation counter $v \leftarrow 0$

Round t **at node** i **:**

observe $x_t^i, y_t^i \subset X \times Y$ with $|x_t^i|$, $|y_t^i| = b$

update f_{t-1}^i using the learning algorithm φ

if $t \bmod b = 0$ **and** $\|f_t^i - r\|^2 > \Delta$ **then**

send f_t^i to coordinator (violation)

At coordinator on violation

let $\mathcal{B}$ be the set of nodes with violation

$v \leftarrow v + |\mathcal{B}|$

if $v = m$ **then** $\mathcal{B} \leftarrow [m]$, $v \leftarrow 0$

while $\mathcal{B} \neq [m]$ **and** $\|\frac{1}{\mathcal{B}} \sum_{i \in \mathcal{B}} f_t^i - r\|^2 > \Delta$ **do**

augment $\mathcal{B}$ by augmentation strategy

receive models from nodes added to $\mathcal{B}$

send model $\bar{f} = \frac{1}{\mathcal{B}} \sum_{i \in \mathcal{B}} f_t^i$ to nodes in $\mathcal{B}$

if $\mathcal{B} = [m]$ also set new reference vector $r \leftarrow \bar{f}$

federated learning setting. We also consider quantization as future work in this study.

In this chapter, we discussed the different elements of federated learning. We introduced the baseline algorithms for this study which discussed the inner workings of machine learning algorithms. Thereafter, we discussed distributed settings and how parallel programming models work. This gives us details of how machine learning is applied in a distributed setting and see how this leads to federated learning. We discuss the details of federated learning and introduce different ways of solving for privacy. This chapter highlights that in order to address the issues of federated learning, statistical heterogeneity needs to tackled in a communication efficient way. In order to tackle statistical heterogeneity, studies suggest that function estimation needs to be updated accordingly. Alternative studies suggest that we could also use oversampling or data exchange methods. It also is of primary importance to find

the balance between computation and communication with negligible impact on error-convergence. FederatedAveraging was one of the key algorithms which followed a periodic averaging method. However, communication is not invested efficiently. This is addressed by dynamic averaging (Kamp et al., 2018) which was one of first studies to introduce the concept of invested communication. This concept is not widely explored and presents an opportunity. There are also no known studies to study the underlying structural properties to inform a dynamic averaging strategy. This is one of the key reasons why we find dynamic averaging particularly interesting. The potential gap that this study aims to explore is to understand the underlying structural properties of the local data at each client to inform a dynamic averaging strategy. This further improves on unbalanced and non-identical data issues that many distributed optimization methods do not address. In the next chapter, we present algorithms that aim to address these potential gaps in dynamic averaging in federated learning.

Chapter 3

Dynamic averaging framework

In this section, we explore the architecture, update rules and the algorithms used for federated optimization. Thereafter, we draw a relationship between the statistical heterogeneity of the data and the communication strategy in a federated learning setting. We see how this improves communication efficiency with negligible impact on model performance. Moreover, we show that it addresses the properties of federated optimization (Section 2.4).

3.1 Architecture

We consider a server/client architecture due to its suitability to federated learning. In a federated learning setting, it is important to detect clients joining or leaving a machine learning training session This includes detecting clients that signal they are ready to performing computations. Thus, the basis of our federated learning algorithm relies on heartbeating (Hintjens, 2013) to detect client states in a distributed setting.

Heartbeating. Heartbeating is the process of determining whether a client is "online" or "offline". "Online" is defined as a client showing signs of life within a certain timeout. We want to know if a client is "online" even if data is not being sent from the client in a training session. The basic pattern follows a ping-pong dialog. The server sends a ping to all clients and the clients respond with a pong within a certain timeout. We follow this pattern for simplicity and we assume that timeouts remain similar across network topologies even though this may not be the case in the real world. We consider the client sending the ping and server responding with a pong in the future to address the different timeouts across network topologies.

Since ping commands are very small in packet size, this does not have significant impact on communication costs. Heartbeating allows for non-blocking operations and allows training to continue while the server waits to receive parameters or when the server does not receive parameters from clients.

Moreover, this allows new clients to be detected dynamically. This ping-pong dialog during a machine learning training session is made possible a heartbeat coordinator. A heartbeat coordinator is run on a separate thread on the server node which collects the locally learned parameters $\mathbf{w}$ to avoid exhausting a single thread. Each client runs a background thread to send a heartbeat back to the coordinator.

Communication patterns. We consider the publish-subscribe and request-reply communication patterns for this framework and discuss how they also enable non-blocking operations. The publish-subscribe (pub-sub) is a data distribution pattern that connects a set of publishers to a set of subscribers. The communication is only one way and publishers are not concerned if a subscriber has received a message or not and messages can be lost. This description tells us this is suitable for distributing mathematical parameters to clients that may join or drop off in a machine learning training session. The request-reply pattern is suitable for collecting data from clients and provides the client alias if needed. In Section 2.2, we discussed that asynchronous updates improve overall communication efficiency but they run the risk of divergence. Therefore, we choose synchronous updates in this study to avoid that risk. The request-reply pattern enables synchronous updates by forcing a block operation. However, due to heartbeating, the system does not come to a halt when a client is no longer active.

Partitioning. In a federated learning setting we have K clients indexed by k. We have P partitions of a given dataset X, so P_k is the partition for client k. To study federated optimization we first need to study the data distribution over all clients. We consider both balanced and unbalanced datasets by simulating different sizes of partitions. Unbalanced datasets can be defined as partitions of unequal size. This is shown by Figure 3.1b. Both datasets are well balanced in this study and obtaining a balanced dataset is trivial. Figure 3.1a demonstrates balanced datasets where each partition is of similar size. To simulate an unbalanced dataset, we randomly pick indices in which to partition the dataset and set a random seed for reproducibility. Figure 3.1b shows how partitioning works for an unbalanced dataset. Moreover, we consider IID, where the data is shuffled to ensure each partition has a similar number of class labels. For non-IID, we order the data by labels and then divide into equal and unequal partitions of varying sizes (McMahan et al., 2016). We ensure that each client has at least two classes to simulate a real-life non-IID scenario. We also make the assumption that each client has enough storage space to hold a partition of data.

(a) Balanced data partition (b) Unbalanced data partitions

Figure 3.1: Partitioning methods. (a) demonstrates balanced data partitioning where we have a similar number of samples in each partition. (b) shows varying sizes of samples at each partition which is prevalent in federated learning.

We note that this architecture does have its limitations and may require a separate coordinator node which detects the heartbeats of thousands of clients. The idea behind this architecture is to provide a good foundation that can be extended easily.

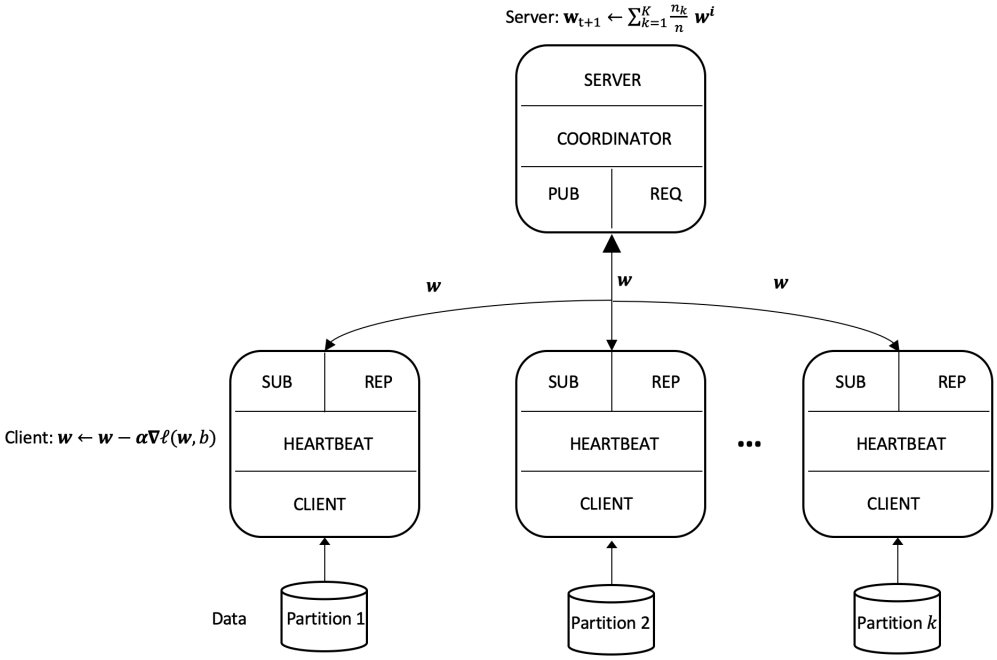

Figure 3.2: The server contains a global model and broadcasts parameters $\mathbf{w}$ to clients using pub-sub communication pattern. Clients perform local updates on their local datasets for a number of local passes. Clients then communicate their local updates back to the server using the request-reply pattern after a number of local passes. The server keeps track of the lifetimes of clients by detecting a client's heartbeat.

3.2 Update rules

In Chapter 2, we introduced the goals of a machine learning algorithm and the update rules required for SGD (Eqn. (2.11)). We provide SGD in the distributed context (*parallel-SGD*) for conciseness. The server initializes the parameters $\mathbf{w}$ and broadcasts to all K clients using the communication pattern described in Section 3.1. Broadcasting these parameters avoids clients having different initial

conditions which results in bad behaviour when averaging models (Goodfellow et al., 2014).

Each client k contains a partition P_k of data. Each client computes local gradients and learns a local update by Eqn (2.11). Communicating these parameters back to the server differs based on the quantization strategy and communication protocol. Upon receiving these parameters from each client, the server aggregates these updates for time step $t+1$ by Eqn. (3.1):

$$\mathbf{w}_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} \mathbf{w}^k, \tag{3.1}$$

where n_k is the number of samples client k holds and n is the total number of samples for a training session.

We provide a base algorithm in Algorithm 2 where we assume that communication back to the server occurs every round. We assume that any encryption will have a small impact on the communication overhead, and this is something to be considered for future work.

Algorithm 2: *Parallel-SGD.* The K clients indexed by k; B is the local minibatch size, E_k is the number of local epochs for client k, and α is the learning rate

Server executes:

Given $\mathbf{w}^{(0)}$

for $t = 1 \rightarrow E$ **do**

The server broadcasts the parameters $\mathbf{w}^{(g)}$ to clients

for *each client* $k \in K$ **do in parallel**

$\mathbf{w}_{t+1}^k \leftarrow \text{ClientUpdate}(k, \mathbf{w}_t)$

$\mathbf{w}_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n^k}{n} \mathbf{w}_{t+1}^k$

ClientUpdate($\mathbf{w}_t$): // *Run on client* k

$\mathcal{B} \leftarrow$ (split P_k into batches of size B)

for *each local* i *from* $1 \rightarrow E_k$ **do**

for *batch* $b \in \mathcal{B}$ **do**

$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla \ell(\mathbf{w}, b)$

return $\mathbf{w}$, $\ell(\mathbf{x}^k, \mathbf{y}^k; \mathbf{w})$

3.3 Algorithms

From Chapter 2, we know that statistical heterogeneity refers to the data being non-IID with each client holding a varying number of samples. To answer the question posed in Chapter 1, we examine how we can understand the heterogeneity of the data without sharing the data with a centralized server.

3.3.1 Dynamic averaging protocol based on SVD

SVD is a method that is commonly used to disentangle the factors of variance underlying the data. Much like how we are able to decompose integers into it's prime factors, we can decompose matrices into their constituent parts to analyze certain properties of the matrix (Goodfellow et al., 2016).

SVD is general purpose matrix factorization method used to decompose an $n \times m$ matrix $\mathbf{M}$ into three distinct matrices:

$$\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T, \tag{3.2}$$

where $\mathbf{\Sigma}$ is a $n \times m$ rectangular diagonal matrix, $\mathbf{U}$ is an $n \times n$-dimensional matrix whose columns are mutually orthonormal and similarly $\mathbf{V}$ is an $m \times m$-dimensional matrix whose columns are mutually orthonormal (Goodfellow et al., 2016).

The elements along the diagonal of $\mathbf{\Sigma}$, σ_i for $i \in \{1, ..., min\{n, m\}\}$ are ordered unique scalars referred to as the singular values. Note that a rectangular $n \times m$-dimensional diagonal matrix consists of two sub-matrices, one square diagonal matrix of size $c \times c$ where $c = min\{n, m\}$ and a matrix of zeros to fill out the rest of the rectangular matrix (Goodfellow et al., 2016). There are j singular values that are large where $j \leq m$. The rest of the singular values approach zero. As a result, terms except the first few can be ignored without losing much of the information. We can then calculate the dimension q for client where $q \leq m$ where q is the dimension of the projected matrix such that we keep 95% of the variance of the original matrix. Therefore q_k represents the dimension that retains 95% of the variance for client k. We do so by calculating the cumulative sum of σ and dividing by the total sum of the singular values σ to obtain the cumulative variance ratio vector $\mathbf{r}$ shown by Eqn. (3.4). It is simple to find the indices of vector $\mathbf{r}$ where $\mathbf{r} \geq 0.95$ by Eqn. (3.5) represented by $\mathbf{r^{0.95}}$. We take the first index or the 0^{th} index of the vector $\mathbf{r^{0.95}}$ to obtain q (Eqn. (3.6)).

$$s = \sum_{j=1}^{m} \sigma_j \tag{3.3}$$

$$\mathbf{r} = \{\frac{\sum_{i=1}^{1} \sigma_i}{s}, \frac{\sum_{i=1}^{2} \sigma_i}{s}, ..., \frac{\sum_{i=1}^{m} \sigma_i}{s}\} \tag{3.4}$$

$$\mathbf{r^{0.95}} = \{i | \mathbf{r}_i \geq 0.95, 0 \leq i \leq m\} \tag{3.5}$$

$$q = \mathbf{r_0^{0.95}} \tag{3.6}$$

The value q is comparable across all clients, and we can get a sense for the structural properties of the local datasets. Higher q values indicate that there are larger singular values and the data contains more significant components. In essence, we invest more communication for clients with higher q values and less communication for clients with lower q values. Thus, we can draw a positive relationship between these q values and communication.

We can represent the corresponding SVD indices $\{q_1, ..., q_k\}$ for K clients by concatenating these q values into a vector $\mathbf{q}$. This is the vector of SVD indices from all clients such that q_k is the SVD index for the 95^{th} percentile for client k. We need to calculate the number of local passes ρ for each client before the client communicates back parameters $\mathbf{w}^k$. The number of local passes for each client can be represented by $\{\rho_1, ..., \rho_k\}$ by the vector $\boldsymbol{\rho}$. In order to calculate ρ, we first normalize $\mathbf{q}$ between a range of $[0, 1]$ using min-max normalization (Patro & Sahu, 2015) to obtain $\mathbf{q'}$ as per Eqn. (3.7). We define $\mathbf{q'}$ as the normalized vector of local passes.

$$\mathbf{q'} = \frac{\mathbf{q} - \min \mathbf{q}}{\max \mathbf{q} - \min \mathbf{q}} \tag{3.7}$$

Recall that the vector $\mathbf{q'}$ consists of values q', where q'_k is the normalized SVD index q for client k. It is now possible to calculate a dynamic number of local passes ρ_k based on q'_k for a client k. Say we have the global maximum number of epochs E. To calculate the total number of communication rounds E_k for a client k, we multiply the number global epochs E by the normalized value q'_k shown by Eqn. (3.8). If the resulting value E_k is 0 because $q'_k = 0$, then $\rho_k = 1$ for client k. This means there is only one communication round for client k. It becomes possible to calculate the number of local passes ρ_k for a client k before communicating parameters back to the server by Eqn. (3.10). If the number of local passes is not a multiple of E, we calculate the epoch $E_k^{(\rho=1)}$ which we switch to $\rho = 1$ whereby the client communicates every local pass by Eqn. (3.11). We have the term $\frac{E}{\rho_k}$ in Eqn. (3.11). This gives us the number of epochs that we use to calculate the number of local passes

ρ_k. We subtract this term $\frac{E}{\rho_k}$ from the communication rounds E_k for client k to get the number of communication rounds we need to communicate every epoch. Thereafter, to calculate the epoch which we switch to $\rho = 1$, we subtract the term $E_k - (\frac{E}{\rho_k})$ from E.

$$E_k = \lceil q'_k \times E \rceil \tag{3.8}$$

$$E_k = \begin{cases} 1 & E_k = 0 \\ E_k & E_k > 0 \end{cases} \tag{3.9}$$

$$\rho_k = \lceil \frac{E}{E_k} \rceil \tag{3.10}$$

$$E_k^{(\rho=1)} = E - (E_k - \lfloor \frac{E}{\rho_k} \rfloor) \tag{3.11}$$

Intuitively speaking, Eqns. (3.8) to (3.11) are needed if the number of local passes ρ_k for client k is not a multiple of the number of epochs E. In order to still communicate the number of calculated epochs E_k for client k, then we calculate the iteration where we switch from the calculated number of local passes ρ_k to communicating every epoch where $\rho = 1$. In some cases, $E_k^{(\rho=1)}$ may be equivalent to E if ρ_k is a multiple of E. This means that we do not need to switch to communicating every iteration.

Eqns. (3.8) to (3.11) demonstrate how we invest the communication efficiently using the normalized SVD indices $\mathbf{q}'$. Once determining the communication protocol, training occurs similarly to Algorithm 2. However, instead of receiving parameter updates from all clients, we only receive updates from the clients we invested communication in for that time step t. Complete pseudo-code is given in Algorithm 3.

We provide the computational complexity of calculating q_k for conciseness. Calculating q_k relies heavily on the SVD computation. The decomposition is performed by the divide and conquer LAPACK routine (Dongarra et al., 2018). This SVD implementation is considered due to use of the Python Numpy implementation. We discuss the implementation details in Section 4.1.

Eqn. 3.8 shows that we are only interested in calculating the singular values. As a result, this reduces the computation complexity of calculating q. The divide and conquer (dgesdd) LAPACK routine

Algorithm 3: Dynamic averaging based on SVD. The K clients indexed by k; B is the local minibatch size, E_k is the number of local epochs for client k, and α is the learning rate

Server executes:

for *each client* $k \in K$ **do in parallel**
$q^k = \text{ClientSVD}(k, \mathbf{w})$

Normalize q values using Eqn. (3.7)

Calculate number of epochs E_k for each client k using Eqn. (3.8)

Calculate number of local passes ρ^k for each client k using Eqn. (3.10)

Calculate epoch where we switch to $\rho = 1$ using Eqn. (3.11)

Given $\mathbf{w}^{(0)}$

// Run parallel-SGD

for $t = 1 \rightarrow E$ **do**
The server broadcasts the parameters $\mathbf{w}^{(g)}$ to clients

$S_t \leftarrow$ (subset of K clients we invested communication in for t)

for *each client* $k \in S_t$ **do in parallel**
$\mathbf{w}_{t+1}^k \leftarrow \text{ClientUpdate}(k, \mathbf{w}_t)$

$\mathbf{w}_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n^k}{n} \mathbf{w}_{t+1}^k$

ClientSVD($\mathbf{w}_t$): *// Run on client k*

Clients compute q using SVD on local dataset by Eqns. (3.3) to (3.6)

return q

ClientUpdate($\mathbf{w}_t$): *// Run on client k*

$\mathcal{B} \leftarrow$ (split P^k into batches of size B)

for *each* t^k *from* $1 \rightarrow E_k$ **do**

for *batch* $b \in \mathcal{B}$ **do**
$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla \ell(\mathbf{w}, b)$

return $\mathbf{w}$, $\ell(\mathbf{x}^k, \mathbf{y}^k; \mathbf{w})$

(Dongarra et al., 2018) has a computational complexity defined by Eqn. 3.12.

$$\text{dgsedd complexity} = \begin{cases} \mathcal{O}(2mn^2 - 2n^3), & m \gg n \\ \mathcal{O}(4mn^2 - \frac{4}{3}n^3), & \text{otherwise} \end{cases}. \tag{3.12}$$

Upon calculating the singular values, we calculate s and $\mathbf{r}$ as per Eqn. 3.3 and 3.4 respectively. The computational complexity of obtaining both s and r is $\mathcal{O}(n)$ since they are linear sum and cumulative sum operations. All that is left to calculate q is to find the index of the singular values such that we keep 95% of the variance. To obtain this index, the computational complexity is also $\mathcal{O}(n)$. The total cost to calculate q is therefore $\mathcal{O}(2mn^2 - 2n^3)$ for $m \gg n$ or $\mathcal{O}(4mn^2 - \frac{4}{3}n^3)$ otherwise.

Similarly, we can construct an alternative communication protocol based on the loss $\ell(\mathbf{x}^k, \mathbf{y}^k; \mathbf{w})$ we receive from each client k. We can consider the loss function for logistic regression for simplicity defined in Chapter 1. We explore this in the next section.

3.3.2 Dynamic averaging based on local loss

We know from Chapter 1, and from general optimization that we minimize/maximize some loss function f. Each client performs local SGD on their local dataset, producing predictions $\hat{y}$. Using a loss function f, where $f = \ell(\mathbf{x}^k, \mathbf{y}^k; w)$, we calculate the epoch loss for each client k. Given the unbalanced nature and diversity of the dataset in a federated learning setting, these loss values may differ substantially. Upon receiving the parameter updates and epoch loss from the clients, an alternative communication protocol is formulated. Similar to how Gradient Boosting (Friedman, 2000) iteratively adds more weight to samples experiencing higher loss, this protocol will invest more communication in clients that have reported back a higher loss.

Once again, we use min-max normalization defined in Eqn. (3.7) to give values between a range of $[0, 1]$ to formulate $\mathbf{q}'$. This time $\mathbf{q}$ represents the loss vector instead of the vector of SVD indices. Similarly to Eqn. (3.8), we calculate the number of communication rounds for a client k. We substitute $E - t$ for E since we want to calculate the number of communication rounds for the remaining number of epochs shown by Eqn. (3.13). Thereafter, we can use Eqn. (3.10) to calculate the number of local passes for a client k.

$$E_k = \lceil q'_k \times (E - t) \rceil \tag{3.13}$$

Once again, if ρ_k is not a multiple of E, we calculate the epoch $E_k^{(\rho=1)}$ which we switch to $\rho = 1$ by Eqn. (3.11). Given that optimization is an iterative process, the number of local passes for a client will be updated multiple times depending on when clients communicate back their loss values. Complete pseudo-code is given in Algorithm 4.

Algorithm 4: Dynamic averaging based on loss. The K clients indexed by k; B is the local minibatch size, E_k is the number of local epochs for client k, and α is the learning rate

Server executes:

Given $\mathbf{w}^{(0)}$

for $t = 1 \rightarrow E$ **do**

The server broadcasts the parameters $\mathbf{w}^{(g)}$ to clients

$S_t \leftarrow$ (subset of K clients we invested communication in for t)

for *each client* $k \in S_t$ **do in parallel**

$\mathbf{w}_{t+1}^k, \ell(\mathbf{x}^k, \mathbf{y}^k; \mathbf{w}^k) = \text{ClientUpdate}(k, \mathbf{w})$

$\mathbf{w}_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n^k}{n} \mathbf{w}_{t+1}^k$

Normalize loss values $\ell(\mathbf{x}^k, \mathbf{y}^k; \mathbf{w}^k)$ using Eqn. (3.7)

Calculate local number of passes ρ_k for each client k using Eqn. (3.10) and (3.11)

ClientUpdate($\mathbf{w}_t$): // *Run on client* k

$\mathcal{B} \leftarrow$ (split P^k into batches of size B)

for *each* t^k *from* $1 \rightarrow E_k$ **do**

for *batch* $b \in \mathcal{B}$ **do**

$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla \ell(\mathbf{w}, b)$

return $\mathbf{w}$, $\ell(\mathbf{x}^k, \mathbf{y}^k; \mathbf{w})$

In Section 2.4, we mentioned that the intuition behind Algorithm 1 and Algorithm 4 are similar. We briefly explain the difference. We mentioned that Algorithm 1 has a divergence threshold that determines how often clients communicate with the server. In other words, clients that violate the divergence threshold will communicate more often. If we look at this from a loss perspective, we see that the intuition of Algorithm 4 is similar. Communication protocols are determined by allowing clients that obtain a higher loss. We see that Algorithm 1 may translate to achieving a greater loss, hence the intuition is similar. The difference is that Algorithm 1 sets a divergence threshold. Algorithm 4 compares loss values obtained from clients and determines invested communication by ranking clients by their normalized loss values.

In this chapter, we presented the architecture suitable for a federated learning environment. In order to keep track of active clients, we choose heartbeating as a mechanism to achieve that. Both algorithms provided in this chapter utilize synchronous updates to avoid running the risk of divergence. In

order to address the properties of federated optimization shown in Section 2.4, we introduce the data partitioning strategies in order to simulate this behaviour. Thereafter, we present the intuition of both Algorithms 4 and 3. Algorithm 3 aims to explore the properties of the underlying local datasets to deduce a dynamic averaging communication strategy. Algorithm 4 aims to use the clients local loss to deduce a dynamic averaging communication strategy. In the next chapter, we look to see how this translates to empirical evaluations.

Chapter 4

Experiments

We are motivated by the idea of invested communication which can address one of the main conundrums of federated learning which is to balance communication and computation. We chose well known datasets to test our experiments. Over 1000 experiments were run, changing a number of parameters which we discuss in this chapter. In this chapter we present the experimental results in a federated learning setting for Algorithms 3 and 4. In Section 4.1, we provide details of the experimental setup in terms of the data and implementation. We explore the effects of a dynamic averaging framework on communication efficiency by analyzing communication metrics of each experiment in Section 4.2. Finally, we analyse the effects of the two dynamic averaging algorithms that focus on invested communication on overall model performance in Section 4.3.

4.1 Experimental details

Data. We present results on the MNIST (LeCun & Cortes, 2010) and Fashion MNIST (Xiao et al., 2017) datasets. Both datasets are of a modest enough size to test the frameworks presented in Chapter 3 given by Algorithms 3 and 4.

(a) MNIST sample (b) Fashion-MNIST sample

Figure 4.1: Dataset samples. (a) MNIST is a handwritten digit recognition dataset. (b) Fashion-MNIST is a MNIST-like fashion product database.

MNIST is a classification dataset containing handwritten digits shown in Figure 4.1a. Each example is a 28×28 grayscale image, associated with a class label from 10 classes with each class label being

represented by a digit $d \in \{0, 1, 2, 3, ...9\}$. The dataset is substantially balanced with each class label having approximately 6000 samples each. The left plot in Figure 4.2 shows this balanced class distribution. We keep the default train test splits provided by LeCun & Cortes (2010) which is given by Table 4.1.

Fashion-MNIST is also a classification dataset which is a MNIST-like fashion product database as shown by Figure 4.1b. It shares the same image size and structure of training and testing splits as MNIST. The dataset consists of a training set of 60000 examples and a test set of 10000 examples shown by Table 4.1. Each example is also a 28×28 grayscale image, associated with a label from 10 classes. Fashion-MNIST is a well balanced dataset with each class label having 6000 samples each as shown by the right plot in Figure 4.2. The default train test splits are used as provided by Xiao et al. (2017) which is given by Table 4.1.

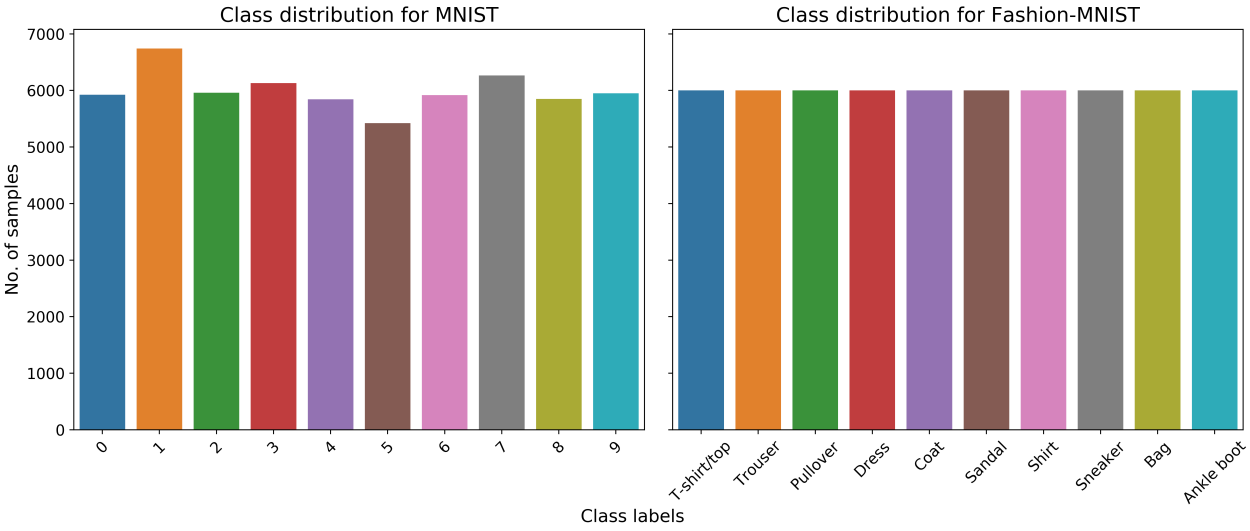

Figure 4.2: Dataset class distributions. The left plot shows MNIST is a well balanced dataset with each class having approximately 6000 samples. The right plot shows a similar trend for Fashion-MNIST which also has 6000 samples per class.

Dataset	Train Samples	Test Samples
MNIST	60000	10000
Fashion-MNIST	60000	10000

Table 4.1: Training and test splits for MNIST and Fashion-MNIST.

Implementation. We make use of the Wits Mathematical Sciences SLURM cluster which has up to 60 nodes each with an i7 7700 CPU, GTX1060 6GB GPU, and 16GB of RAM. We only

considered homogenous devices in this study. This is due to the SLURM resource manager which equally distributes the workload. As a result, update rates are similar since hardware specifications of the nodes in the cluster are similar. We implement the dynamic averaging framework in Tensorflow (Abadi et al., 2016a) using ZeroMQ (Hintjens, 2013) as the message passing library. ZeroMQ is a high performance asynchronous messaging library that supports the communication patterns for the frameworks presented in this study as discussed in Section 3.1. These frameworks provide Python interfaces and hence Python is used for this research study. This allows for rapid prototyping of concepts and flexibility of communication patterns. The frameworks have multi-language support in both Python and Java. Consequently, the framework can be translated to android devices. We implemented serialization using Google's protocol buffers. This is native to Tensorflow (Abadi et al., 2016a) and therefore this framework has the ability to be deployed to android devices.

Setup. We handle this convex problem of classification using logistic regression described by Eqn. (2.2) due to it's simplicity. Table 4.2 displays the layer type, the shape and as a result the number of parameters in a logistic regression model architecture. The choice of this architecture is self-explanatory since this architecture is standard.

Layer Type	**Output shape**	**No. of parameters**
Flatten	(784)	0
Dense	(10)	7 850
Total		7 850

Table 4.2: The model architecture of logistic regression. The model summary is made available by the Keras library in Tensorflow.

Moreover, we explore the non-convex problem of classification by implementing a simple neural network and a simple convolutional neural network. Tables 4.3 and 4.4 show the model architectures for NNs and CNNs respectively. The choice of the one-hidden layer neural network was to introduce a non-convex model space and test the different algorithms under an entry level non-convex model. A similar explanation is for the convolutional neural network architecture. A common entry-level convolutional neural network architecture is presented to introduce more complexity than the simple neural network architecture. We see that there is only one convolution layer and max pooling layer. This architecture is common for basic convolutional neural networks and hence suitable for this study.

By covering these different types of models, we provide good coverage of different model spaces. We

Layer Type	Output shape	No. of parameters
Flatten	(784)	0
Dense	(128)	100 480
Dense	(10)	1290
Total		101 770

Table 4.3: The model architecture of a simple neural network. This neural network has a single hidden layers. The model summary is made available by the Keras library in Tensorflow.

Layer Type	Output shape	No. of parameters
Conv2d	$(26, 26, 32)$	320
MaxPooling2D	$(13, 13, 32)$	0
Flatten	(5408)	0
Dense	(128)	692 352
Dropout	(128)	0
Dense	(10)	1290
Total		693 962

Table 4.4: The model architecture for a simple convolutional neural network. The model summary is made available by the Keras library in Tensorflow.

do not consider more complex architectures to allow us to focus on the elements related to federated learning which we discuss in this section. That being said, by utilizing the entry level of complex models will show that the techniques discussed in this study can be translated to more complex models.

Figure 2.3 showed how logistic regression can be represented as a neural network. We introduced activation functions in the corresponding section. The activation function that is utilized for logistic regression is the sigmoid function. This is also applied for the simple neural network. For CNNs, we considered the ReLU function. This is an alternative activation function. They can be defined by Eqn. (4.1). ReLU functions are easier to optimize and have become a common activation to use in a neural network architecture.

$$g(z) = max\{0, z\} \tag{4.1}$$

For all models for MNIST, we fix the learning rate $\alpha = 0.01$ for SGD and 0.001 for Adam respectively. We fix the batch size $B = 128$ for a number of epochs $E = 100$ for logistic regression and $E = 50$

for both neural network architectures. We choose a lower number of epochs for both neural network architectures to avoid unnecessary epochs. We observed that there was not much improvement after 50 epochs. Similarly, for Fashion-MNIST we fix the learning rate, batch size and number of epochs to the same values. For both datasets, we implement the base *parallel-SGD* algorithm (Algorithm 2), FederatedAveraging (Abadi et al., 2016b) and dynamic averaging (Kamp et al., 2018) to draw a fair comparison.

For each experiment, we change the number of clients, the dataset, the optimizer, the communication protocol, the number of local passes ρ, and the different partition strategies discussed in Section 3.1. We run the FederatedAveraging and dynamic averaging methods on K clients where $K \in \{7, 15, 23, 31\}$. The numbers in this set are odd since one node acts as the server and the rest as clients. So if we consider the total number of nodes used in the training session, it would be $\{8, 16, 24, 32\}$. The highest number of nodes participating was capped at 32 nodes due to system constraints of the cluster at the time of running these experiments. Even though federated learning may scale beyond these numbers, this still allows us to test the scalability of the dynamic averaging algorithms. Our architecture allows for dynamic clients and when starting a training session we set an initial timeout of 30 seconds to allow for clients to join. However, clients can also join after a training session has begun. This is more of a practical consideration and something that we do not measure in the empirical evaluation since the studies that we compare to do not explicitly present this.

We study four ways of partitioning the data since these elements are key properties of federated optimization. We have **IID**, where the data is shuffled, such that each partition is representative of the entire dataset. Then, we partition the data across K clients where each client receives $60000/K$ samples. We also have **non-IID**. We first sort the data by digit label, and partition the data across K clients where each client receives $60000/K$ samples. As discussed in Section 3.1, we ensure that each client has at least two classes in it's partition. We then study balanced and unbalanced datasets. Achieving a **balanced** dataset is trivial. We partition the data across K clients where each client receives $60000/K$ examples. We achieve an **unbalanced** dataset by simulating different sizes of partitions for each client when mapping the dataset. We randomly pick indices in which to partition the dataset and set a random seed for reproducibility. Note that setting a different seed impacts the model performance since this changes the partition sizes and potentially the structure of the underlying data. However, setting the seed allows for comparable experiments.

To make a fair comparison, we track two key metrics. Firstly, we track the communication metrics, more specifically the accumulated packet size sent between clients and server. A packet is the data sent over a network. By measuring the packet size, we get an indication of how much communication overhead we encounter for a particular experiment. We also look at how this translates to communication rates per client. Communication rate can be defined as how often a client communicates over a number of epochs E in a federated learning training session. The average communication rate is an average across clients to be able to compare different experiments. Secondly, we track model performance metrics such as accuracy and how well the model generalizes in a federated learning setting. We want to determine if the algorithms have a positive effect on model performance or negligible impact on model performance. It is important to highlight that a seed was set for reproducibility. Therefore, the metrics provided in the upcoming sections for each scenario are completely reproducible and no error bars are visible.

4.2 Investing communication efficiently

Consider FederatedAveraging (McMahan et al., 2016) with the number of local passes $\rho > 1$ and $\rho \in \{1, 10, 20, 50, 100\}$. The number of communication rounds reduces with an increasing number of local passes ρ and hence the accumulated packet size reduces. If we reduce the number of of communication rounds by a factor of x then the accumulated packet size will reduce by a factor of x.

Let us consider a simple example to illustrate how the accumulated packet size is calculated. We consider a logistic regression model for this example and the MNIST dataset. We know that MNIST contains 28×28 grayscale images and 10 classes. Thus, our model parameters $\mathbf{w}$ is a matrix of size 784×10. If we translate this to bytes, then we consider the data type of this matrix. Let's assume this is a matrix is made up of 32-bit floating point values. Recall that 1 byte is 8 bits. So a float-32 matrix of size 784×10 is of size $784 \times 10 \times 4 = 31360$ bytes. We want to train our model for 100 epochs. We set the number of local passes $\rho = 1$. Therefore, the number of communication rounds equates to 100. For a single client, the total packet size communicated for one way communication in megabytes is $31360 \times 100/1024/1024 = 2.99$MB. If we have 10 clients, then the overall packet size is $2.99 \times 10 = 29.9$MB.

Let us consider the three models that we have implemented in this study. Namely, logistic regression, a neural network and a convolutional neural network. Due to the different model architectures, they will each have different number of parameters sent to and from server and client. Moreover, we have

different datasets and different optimizers. We will therefore break down the packet size analysis by model, dataset and optimizer.

4.2.1 Logistic regression (LR)

In Section 3.3.1 and 3.3.2, we discuss the algorithms to reduce overall packet size. Let us consider logistic regression first. We refer to Figures 4.3, 4.7, 4.5, 4.9 for analysis. In the rest of this chapter, we refer to Algorithm 1 as DynAvg. In each Figure, we refer to Algorithm 3 as DynAvg SVD and Algorithm 4 as DynAvg Loss.

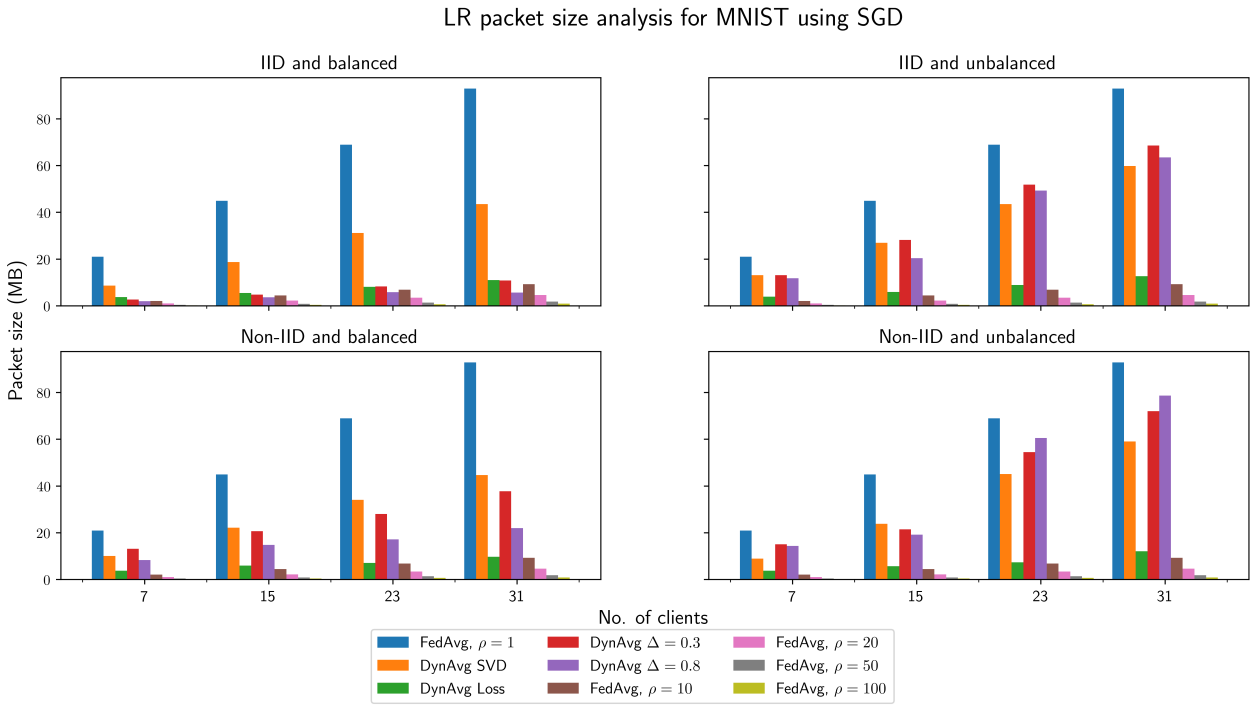

Figure 4.3: Logistic regression packet size analysis for different communication protocols across a different number of clients on MNIST using SGD: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

MNIST and SGD. We consider each figure one by one to highlight the differences and similarities. Figure 4.3 demonstrates a significant decrease in packet size for FederatedAveraging when $\rho \in \{10, 20, 50, 100\}$ relative to FederatedAveraging when $\rho = 1$ for IID and balanced datasets. This pattern is the same across all different partitioning strategies. This is expected for a periodic averaging technique since it is not dynamic.

DynAvg $\Delta \in \{0.3, 0.8, 2.2, 2.8\}$ (Kamp et al., 2018) differs across the different partition strategies in terms of packet size. For IID and balanced datasets (upper-left plot) for SGD, DynAvg $\Delta \in \{0.3, 0.8\}$

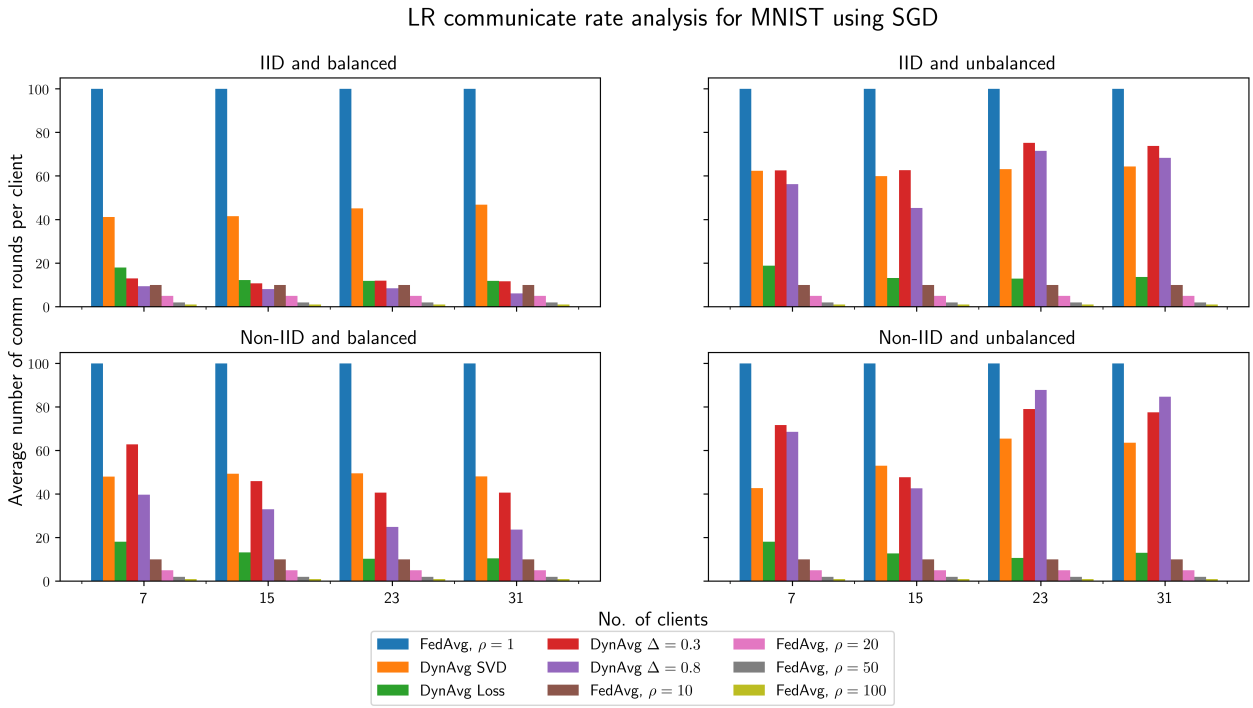

Figure 4.4: Logistic regression communication rate analysis across a different number of clients on MNIST using SGD for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

is comparable to FederatedAveraging $\rho > 1$. This is since the data distributions across clients will be similar. Therefore, when each client learns, they learn at a similar rate and thus reach some dynamic averaging threshold at a similar epoch. This results in communicating periodically and is thus comparable to FederatedAveraging $\rho > 1$. Figure 4.4 confirms this as the average number of communication rounds remains almost identical when increasing the number of clients. Algorithm 3 communicates approximately 2.5x less than FederatedAveraging $\rho = 1$ but accumulated packet size is greater than the other federated learning techniques. If we inspect Algorithm 4, we see that the accumulated packet size is comparable to DynAvg $\Delta \in \{0.3, 0.8\}$ as well as FederatedAveraging $\rho > 1$.

If we consider non-IID and balanced datasets (bottom-left plot), then we notice an increase in communication for DynAvg $\Delta \in \{0.3, 0.8\}$ also shown by Figure 4.4. This is due to the nature of non-IID data. Each local model violates the threshold more often upon synchronization since the local models are vastly different. The number of communication rounds slightly decreases across the number of clients. However, we still see the accumulated packet size increase with the number of clients. This may suggest that this strategy attempts to invest communication efficiently with the increase in the number of clients. Expectedly, FederatedAveraging across all different ρ remains similar. Algorithm 3

remains similar for all balanced datasets. This means that after normalizing SVD indices that keep 95% of the variance we get similar communication intervals for balanced datasets. This suggests that for non-IID datasets, Algorithm 3 struggles to portray an accurate representation. This highlights that there may exist a shortfall of SVD for purely non-IID settings which we discuss in Section 4.3. Algorithm 4 also remains similar for balanced datasets. This suggests that for non-IID datasets that loss across clients may be similar. The global model may be different to local models in a similar manner due to each client having different class labels.

For IID and unbalanced datasets (upper-right plot) there is an increase in communication for dynamic averaging techniques shown by Figure 4.4 resulting in greater accumulated packet size. Unbalanced datasets mean that certain clients may have less data. This may lead to certain clients communicating more often balanced datasets. For DynAvg techniques, clients that violate the divergence condition communicate more often. For Algorithm 3, clients who lose the least information when calculating SVD communicate more often. Algorithm 4 give weight to clients experiencing a higher loss but there appears to be a slight increase in communication compared to the other partitioning strategies. Given the logic of these algorithms, we can see how communication increases. It is also important to note that the normalization step shown by Eqn. (3.7) impacts how different the communication intervals are. We considered min-max normalization for both Algorithms 3 and 4. This means that clients communication intervals may be enlarged. We first discuss the model performance in Section 4.3 and then discuss this.

We see similar patterns for non-IID and unbalanced datasets (bottom-right plot) in terms of accumulated packet size. DynAvg techniques are comparable with Algorithm 3. Algorithm 4 remains similar in comparison to other partitioning strategies. In terms of average communicate rate we notice that for non-IID and unbalanced datasets that we encounter slightly more regular communication in comparison to the other partitioning strategies. For dynamic averaging techniques such as Algorithm 3, Algorithm 4 and both DynAvg techniques, this is expected. If we look at non-IID for balanced datasets, we notice an increase in communication. The same can be said for unbalanced and IID datasets. Both of these partitioning strategies cause the data to be statistically heterogeneous even though they are different methods. Therefore, this causes the results for DynAvg techniques to be somewhat similar because DynAvg responds to statistical heterogeneity and does not care about non-IID or unbalanced specifically. However, we notice that there is greater communication for DynAvg techniques for 23 clients and 31 clients for non-IID and unbalanced datasets. The increase may be due to the fact that as

the number of clients increases, the data partition size that each client has decreases. This means that when considering non-IID and unbalanced datasets, each partition has fewer number of samples. Given how we have set up our experiment as shown by Figure 3.1 in Section 3.1 in terms of partitioning, this means that each client has fewer number of class labels. This leads to greater divergence of the violation threshold for DynAvg techniques and thus resulting in increased communication.

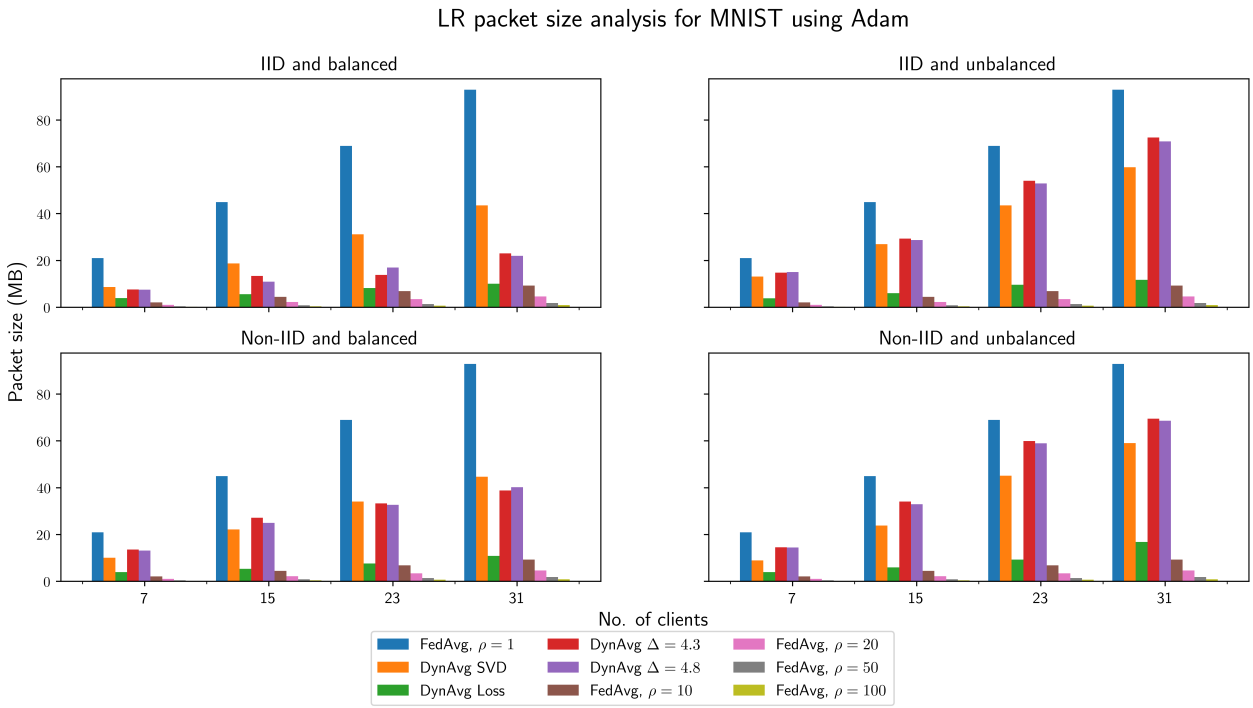

Figure 4.5: Logistic regression packet size analysis across a different number of clients on MNIST using SGD for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

MNIST and Adam. We see similar trends for FederatedAveraging $\rho \in \{1, 10, 20, 50, 100\}$ and Algorithms 3 and 4 across all partitioning strategies in Figure 4.5. This is not the case for DynAvg $\Delta \in \{2.2, 2.8\}$ using Adam. We discovered that when using Adam as an optimizer, setting the violation threshold Δ needs careful consideration. This is because having a faster learner such as Adam or a higher learning rate means that the difference between the reference model and the local models results in large divergence. This leads to clients exceeding the divergence threshold often which results in more regular communication. This may not necessarily mean that we are investing communication efficiently. We therefore require additional hyperparameter tuning which is not ideal when considering the number of hyperparameters needed to tune for more complex models. As a result, we see greater accumulated packet sizes for DynAvg in Figure 4.5 for all partitioning strategies. For IID and balanced

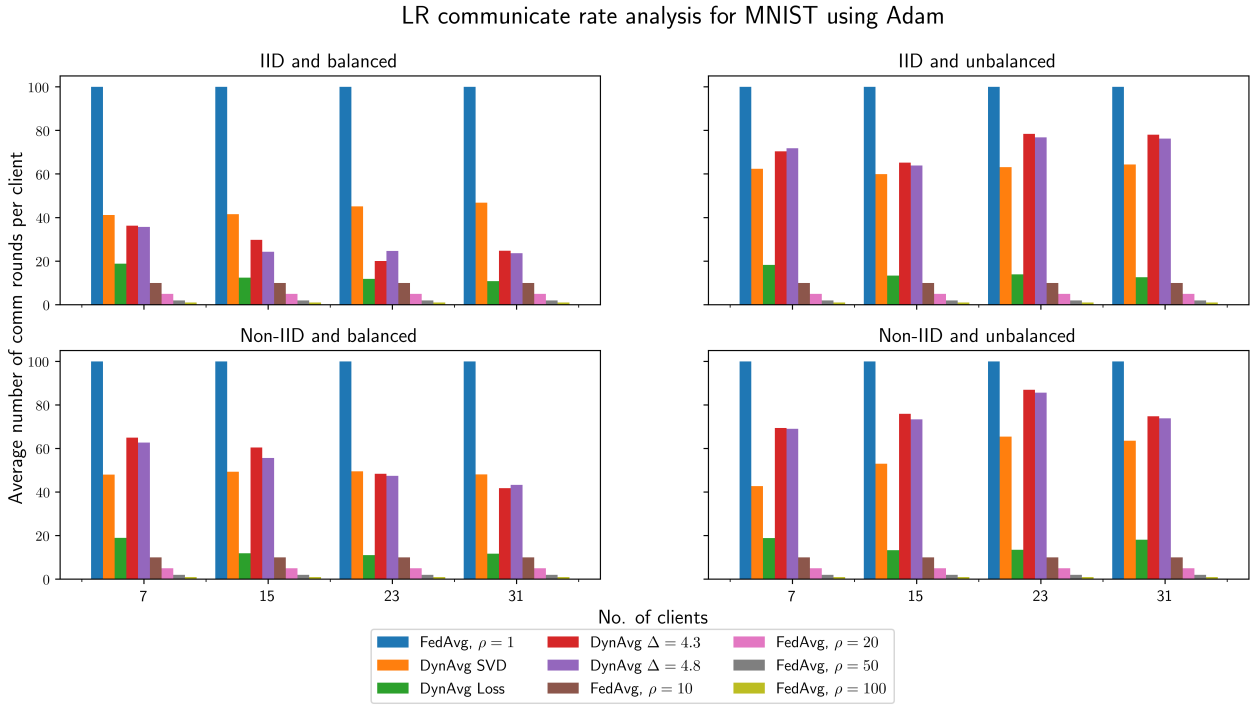

Figure 4.6: Logistic regression communication rate analysis across a different number of clients on MNIST using Adam for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

datasets, DynAvg $\Delta \in \{2.2, 2.8\}$ is comparable to Algorithm 3. For the other three partitioning strategies, DynAvg becomes more comparable to FederatedAveraging $\rho = 1$.

This trend is constant throughout all partitioning strategies when using Adam as an optimizer. This highlights that setting a threshold Δ may not be as trivial as it seems. Given that the dataset remains the same, the accumulated packet size and average communication rate remain the same for Algorithm 3. Algorithm 4 differs slightly but there are no major changes. This is since the client loss relative to each other is similar given the data distributions are the same for SGD. Since the only variable changed is the optimizer and all clients use the same optimizer, this does not impact the amount of communication sent significantly.

Fashion-MNIST and SGD. In terms of communication metrics, the trends from Fashion-MNIST using SGD experiments are similar to that of MNIST using SGD. For IID and balanced datasets, DynAvg $\Delta \in \{0.3, 0.8\}$ is once again comparable to FederatedAveraging $\rho > 1$. There is linear growth in the accumulated packet size for FederatedAveraging $\rho = 1$. We notice that this is not the case for Algorithm 3. There appears to be a plateau suggesting that this strategy aims to invest communication efficiently.

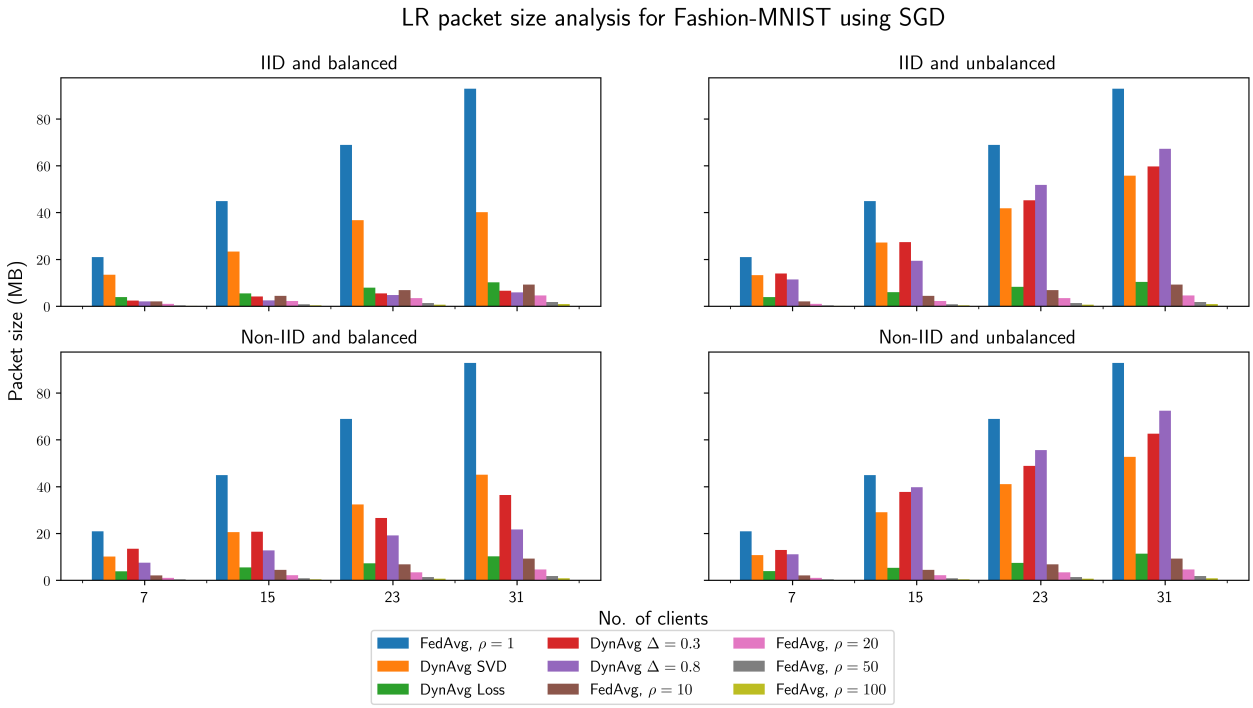

Figure 4.7: Logistic regression packet size analysis across a different number of clients on Fashion-MNIST using SGD for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss

For non-IID and balanced datasets, there is a linear increase in the accumulated packet size for all strategies but is more gentle in comparison to FederatedAveraging $\rho = 1$. This is similar to the MNIST dataset using SGD. This trend also appears for both IID and balanced datasets and non-IID and balanced datasets. Much like the MNIST dataset, there is greater communication for all unbalanced partitioning strategies shown by Figure 4.4.

Fashion-MNIST and Adam. The difficulty of setting the threshold Δ for DynAvg is once again evident for Fashion-MNIST shown in Figures 4.5 and 4.6. DynAvg $\Delta = 0.3$ is comparable to Algorithm 3 for IID and balanced datasets. We observed this for MNIST as well.

We can see how this impacts the accumulated packet size and average communication rate for the other three partitioning strategies for DynAvg. DynAvg for $\Delta \in \{2.2, 2.8\}$ communicates more than all the other algorithms besides FederatedAveraging $\rho = 1$. Moreover, we see that for non-IID and unbalanced datasets, DynAvg becomes more comparable to FederatedAveraging $\rho = 1$ in terms of communication. This is not ideal, as we have not invested communication efficiently as a result of setting this threshold incorrectly.

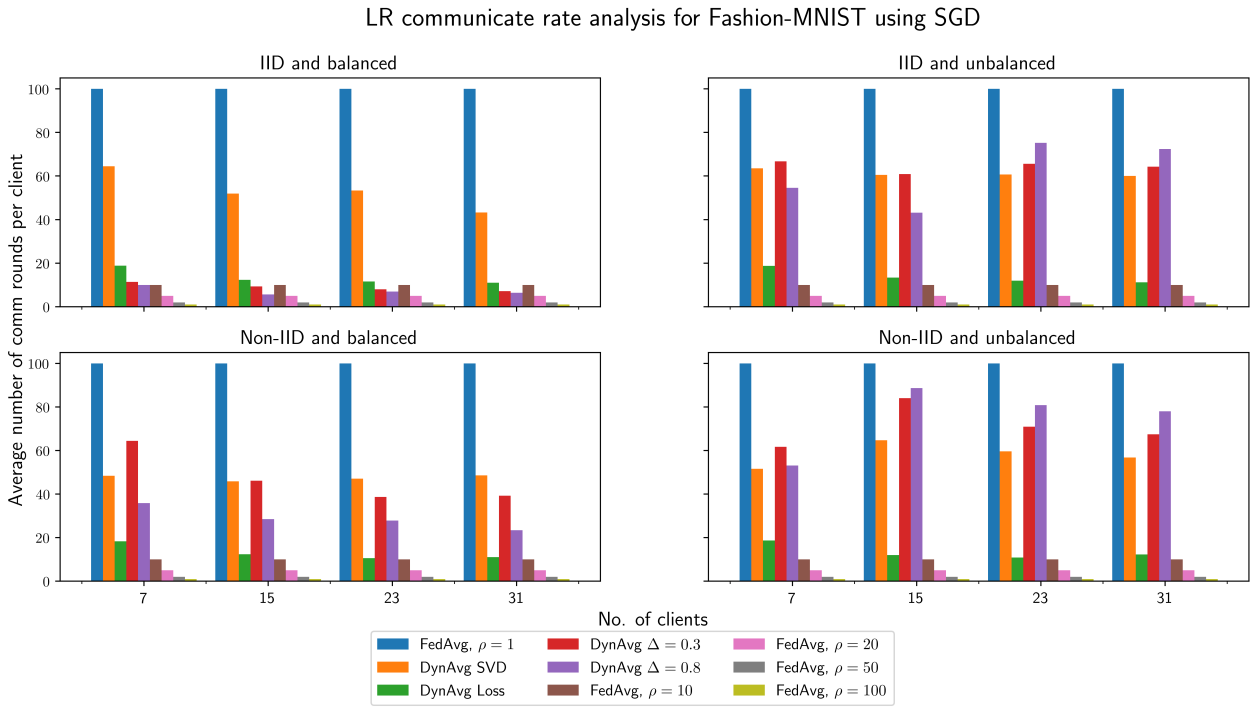

Figure 4.8: Logistic regression communication rate analysis across a different number of clients on Fashion-MNIST using SGD for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

4.2.2 Simple neural network

We now consider a simple neural network to inspect communication metrics. We refer to Figures 4.11, 4.15, 4.13, 4.17 for analysis.

MNIST and SGD. Once again we consider each dataset and optimizer one at a time. Neural networks consist of additional complexity in comparison to logistic regression models and hence a greater number of model parameters. This is shown by Table 4.3. Figure 4.11 demonstrates this increase in the number of parameters if we compare to Figure 4.3.

If we consider IID and balanced datasets we notice slightly different behaviour for Algorithm 4 for neural networks. For logistic regression, Algorithm 4 was comparable to DynAvg and also FederatedAveraging $\rho > 1$. However, Figures 4.11 and 4.16 show that Algorithm 4 has increased communication for this particular partitioning strategy. This may due to the fact that we trained our neural network for less epochs. As discussed in Section 4.1, we chose a lower number of epochs for neural networks since they achieved convergence quicker for these particular datasets. The other algorithms show similar trends

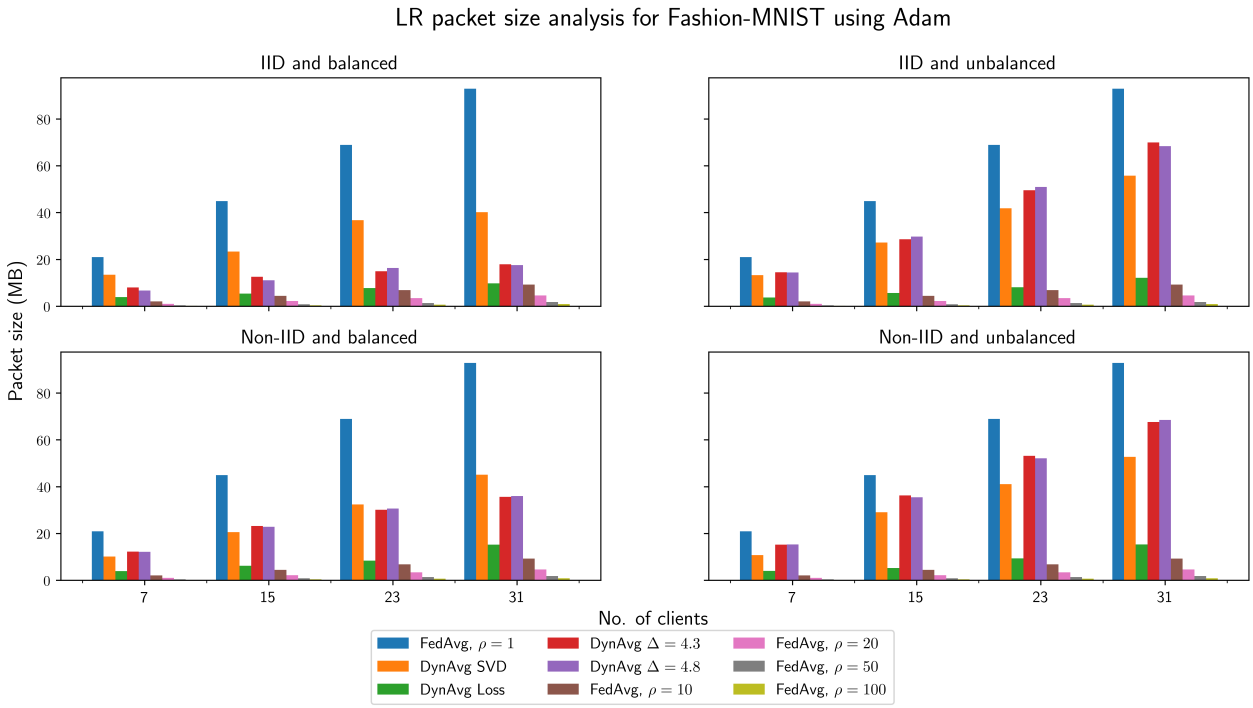

Figure 4.9: Logistic regression packet size analysis across a different number of clients on MNIST using Adam for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

as they did for logistic regression shown by Figure 4.3.

Algorithm 3 proves to be similar in terms of communication strategies for both balanced dataset experiments (left-hand-side plots). We discuss this more in detail in Section 4.3. Similarly to logistic regression, we see that DynAvg $\Delta \in \{0.3, 0.8, 2.2, 2.8\}$ (Kamp et al., 2018) differs across the different partition strategies in terms of packet size. We see this for the non-IID and balanced dataset experiment.

For techniques that employ dynamic averaging which rely on the data, communication changes significantly. The more complex the federated learning environment, the more work each client is forced to do in order to aim to invest communication efficiently. We refer to federated learning environment with high complexity as a distributed environment that shows all properties of federated optimization (Section 2.4). This is expected and is seen for both unbalanced dataset cases. When synchronizing models, we result in an average of all local models. Each client then received the updated global model. This may lead to the updated local learner diverging far often given the nature of it's local dataset. Thus, leading to more communication.

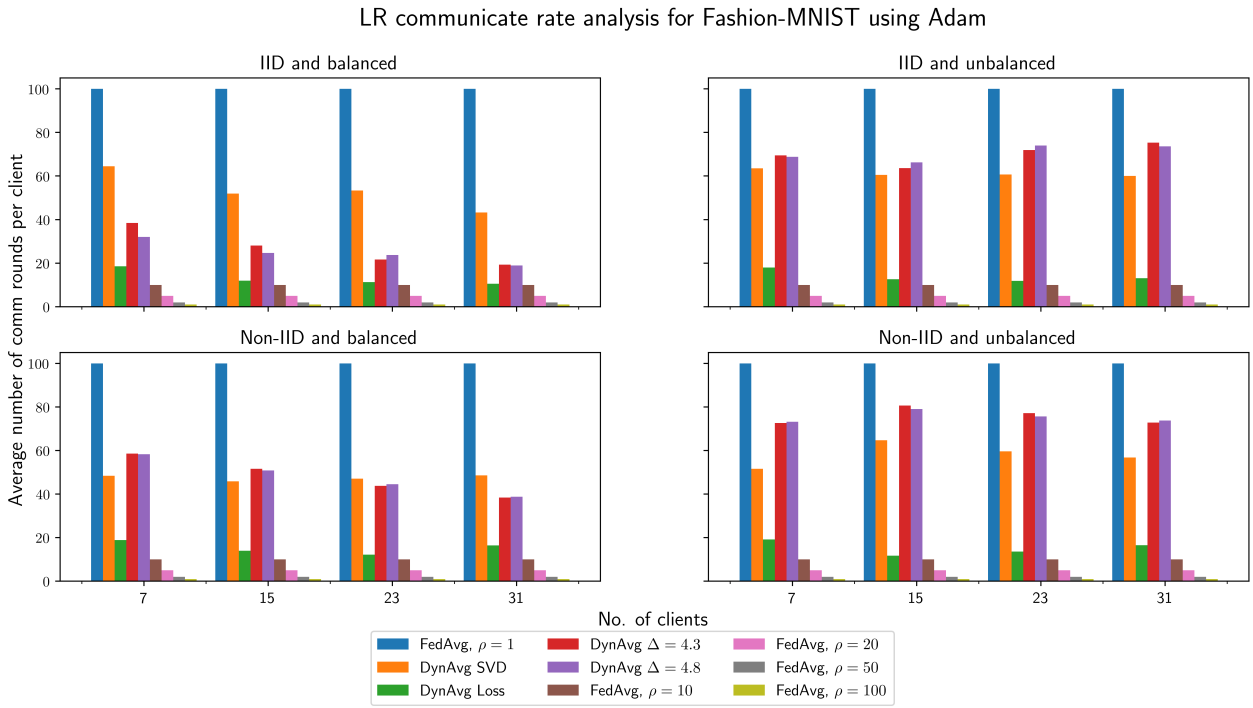

Figure 4.10: Logistic regression communication rate analysis across a different number of clients on Fashion-MNIST using Adam for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

MNIST and Adam. FederatedAveraging for $\rho \in \{1, 10, 50\}$ remains consistent in terms of communication across all partitioning strategies. We also see this consistency in both algorithms 3 and 4 across all partitioning strategies in Figures 4.13 and 4.14. Once again, this is not the case for DynAvg $\Delta \in \{2.2, 2.8\}$ using Adam. There is a significant difference in the amount of communication when compared to SGD and logistic regression. This once again highlights that setting the divergence threshold Δ is important when using Adam as optimizer. For IID and balanced datasets DynAvg becomes comparable to FederatedAveraging $\rho = 1$ suggesting we aren't investing communication efficiently.

This trend is constant throughout all partitioning strategies when using Adam as an optimizer. Algorithms 3 and 4 remain similar to SGD experiments on MNIST. We've mentioned that this is due to the dataset remaining the same. There is a continued trend where dynamic averaging techniques increase the amount of communication for all unbalanced dataset experiments. However, we note that the communication rate for DynAvg is significantly higher for Adam than for logistic regression. We have already discussed this when we considered the results for MNIST.

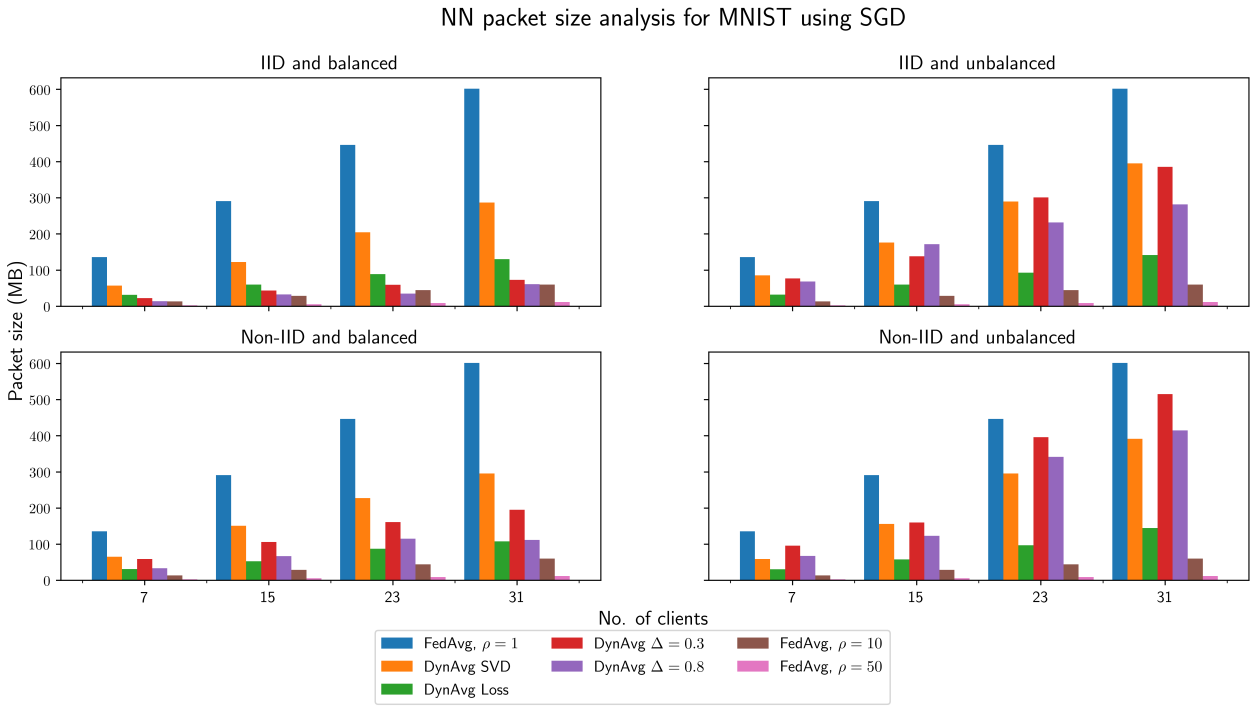

Figure 4.11: NN packet size analysis across a different number of clients on MNIST using SGD for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

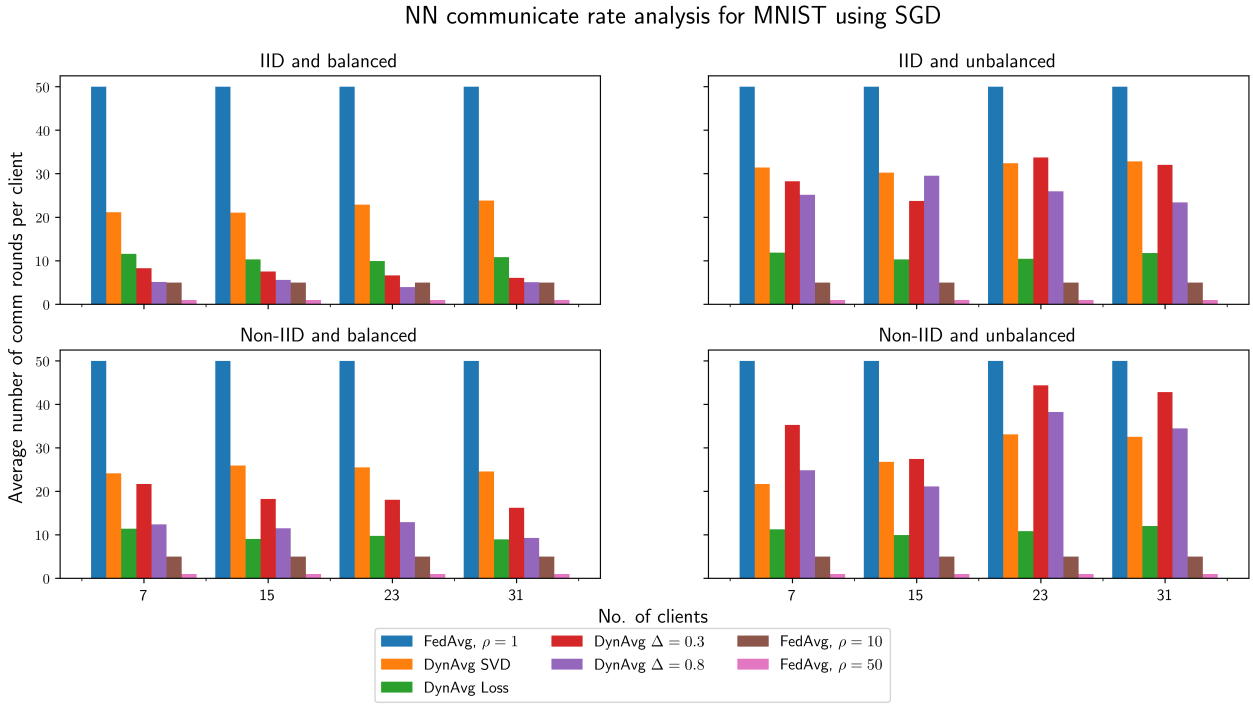

Figure 4.12: NN communication rate analysis across a different number of clients on MNIST using SGD for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

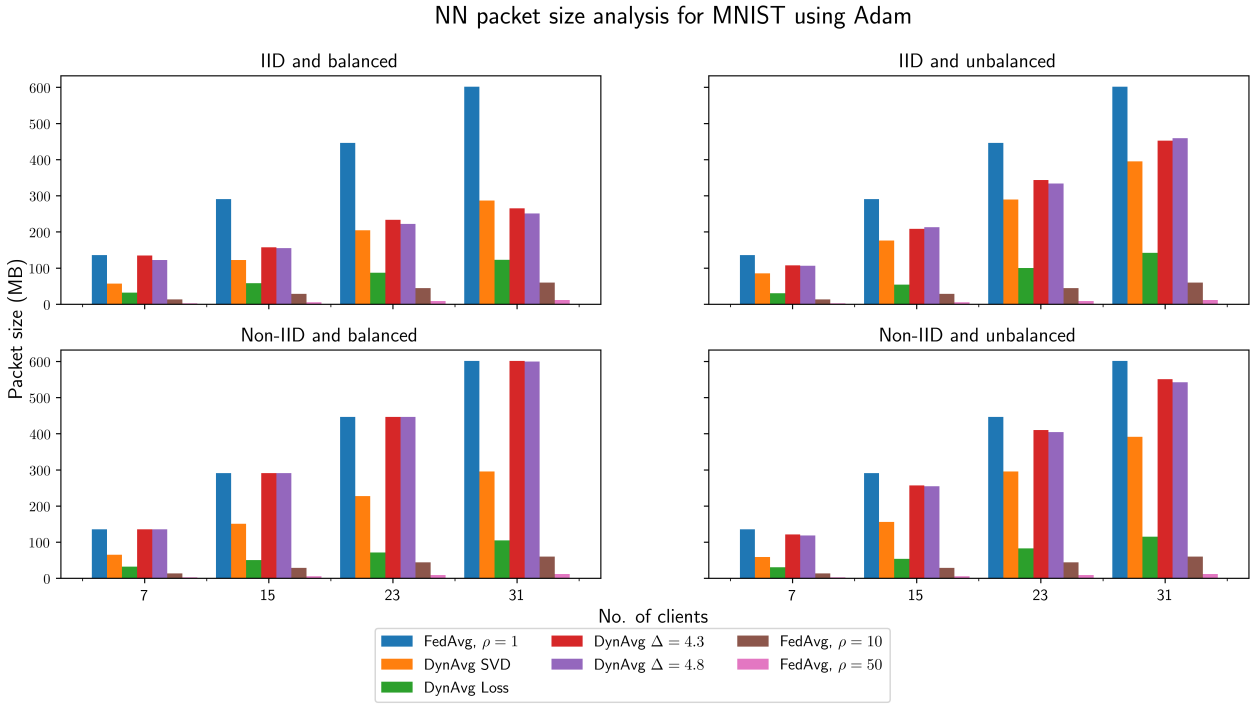

Figure 4.13: NN packet size analysis across a different number of clients on MNIST using SGD for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

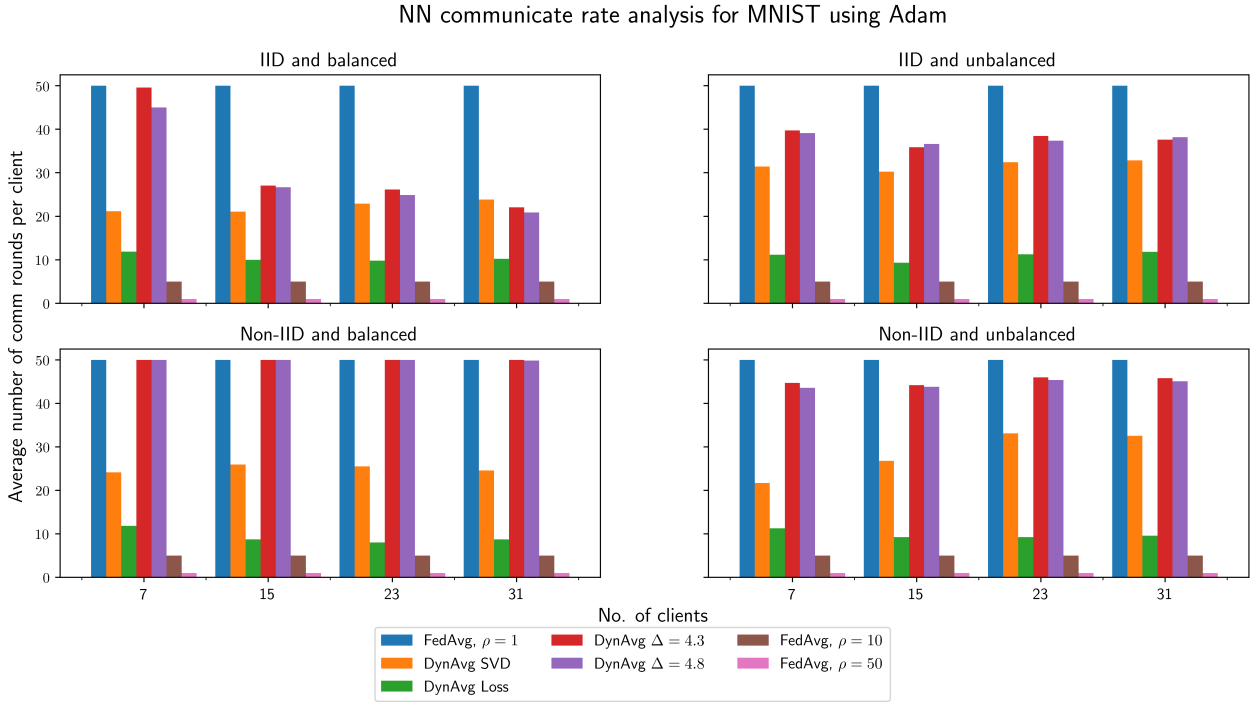

Figure 4.14: NN communication rate analysis across a different number of clients on MNIST using Adam for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

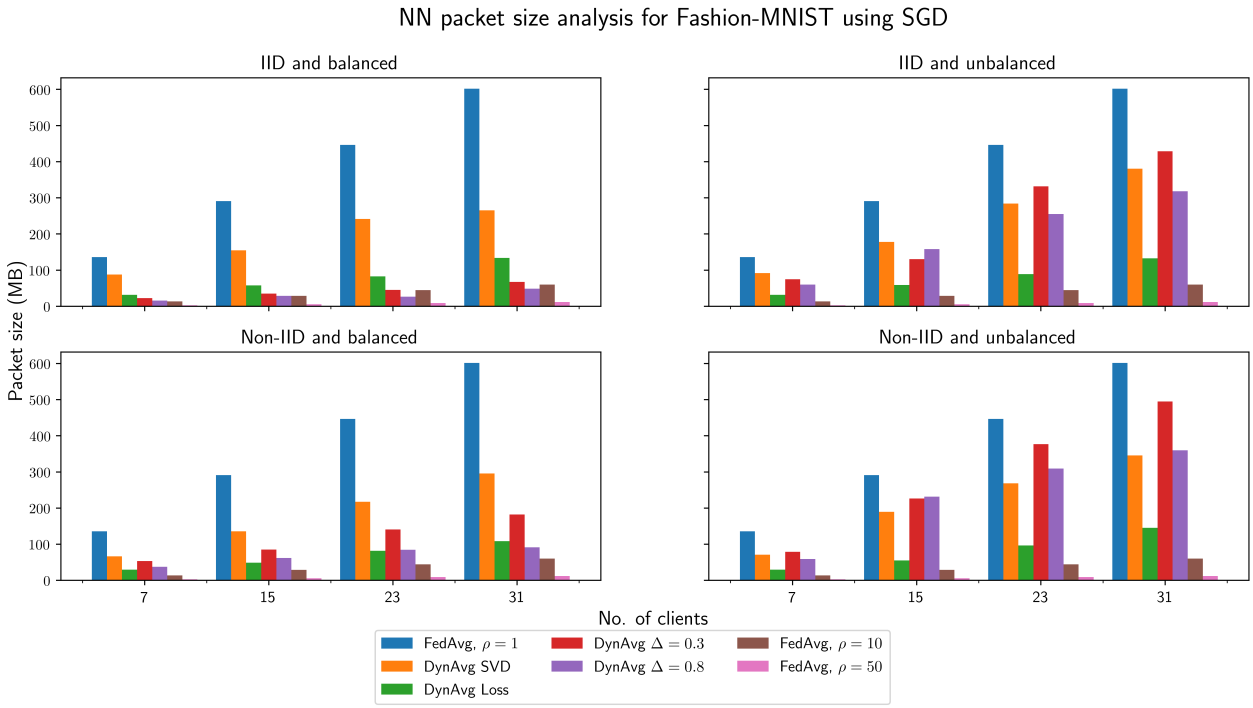

Figure 4.15: NN packet size analysis across a different number of clients on Fashion-MNIST using SGD for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

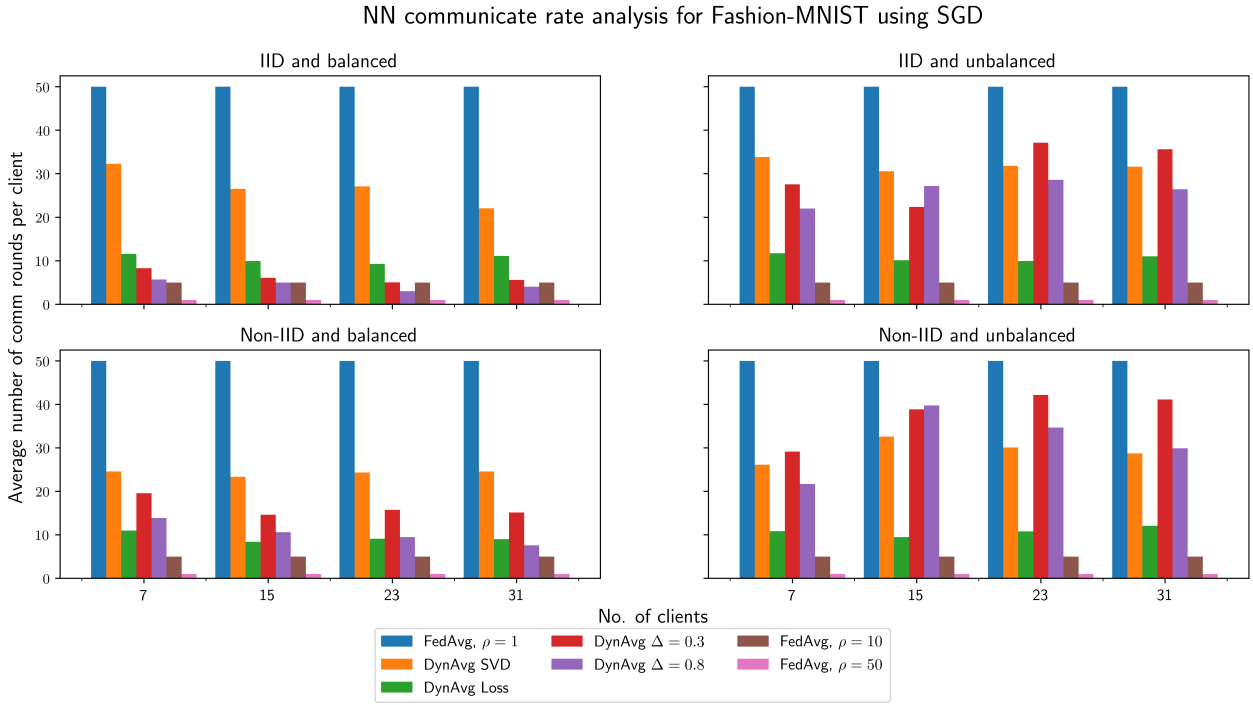

Figure 4.16: NN communication rate analysis across a different number of clients on Fashion-MNIST using SGD for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

Fashion-MNIST and SGD. Communication metric trends are similar for Fashion-MNIST to MNIST for SGD for neural networks. We see a similar linear increase in accumulated packet size as well as an increase in accumulated packet size for unbalanced dataset experiments. Similarly to logistic regression, there appears to be a plateau for IID and balanced datasets.

For non-IID and balanced datasets, Algorithms 3 and 4 show similar communication strategies in comparison to IID and balanced datasets. There is an increase in communication for DynAvg $\Delta \in \{0.3, 0.8\}$. Much like the MNIST dataset, there is greater communication for dynamic averaging strategies for all unbalanced partitioning strategies shown by Figure 4.12.

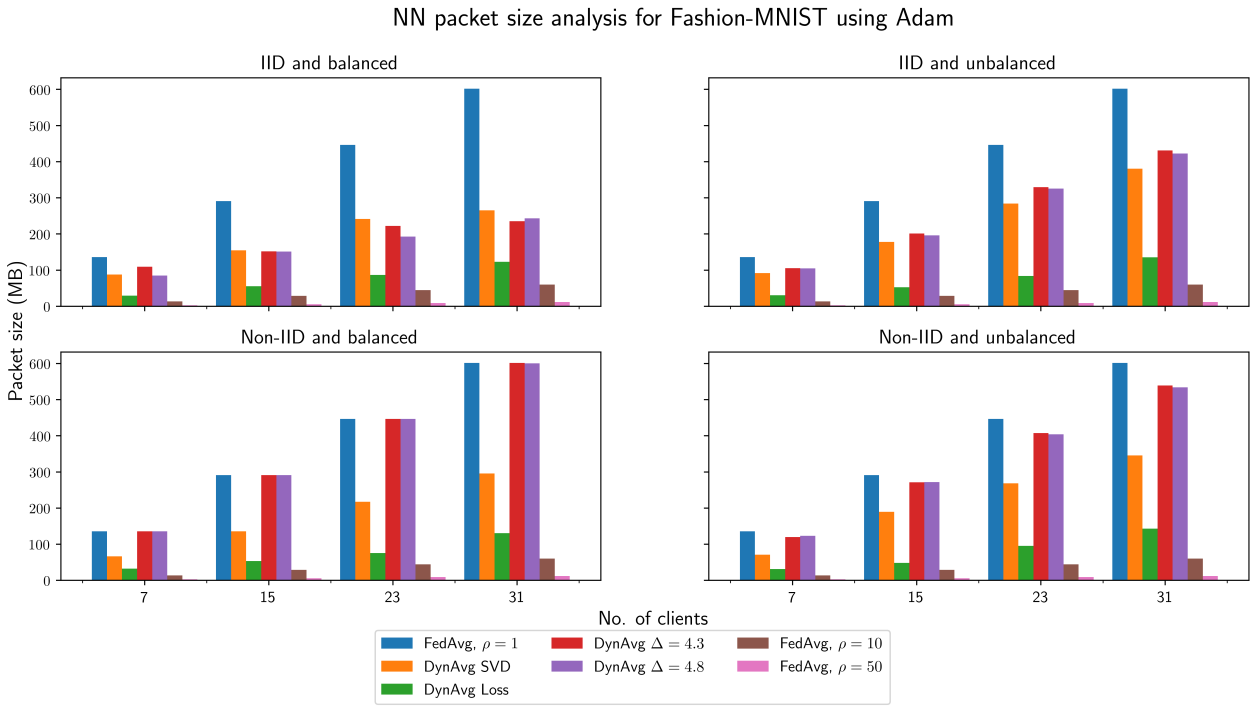

Figure 4.17: NN packet size analysis across a different number of clients on Fashion-MNIST using Adam for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

Fashion-MNIST and Adam. We achieve similar communication results for Fashion-MNIST using Adam as we did for MNIST. DynAvg once again is comparable to FederatedAveraging $\rho = 1$. The other algorithms remain similar in their communication strategies.

4.2.3 Simple convolutional neural network

In Section 2.1, we mentioned the shortfalls of neural networks when we introduced convolutional neural networks. Table 4.4 shows that we have an increased number of model parameters. This allows

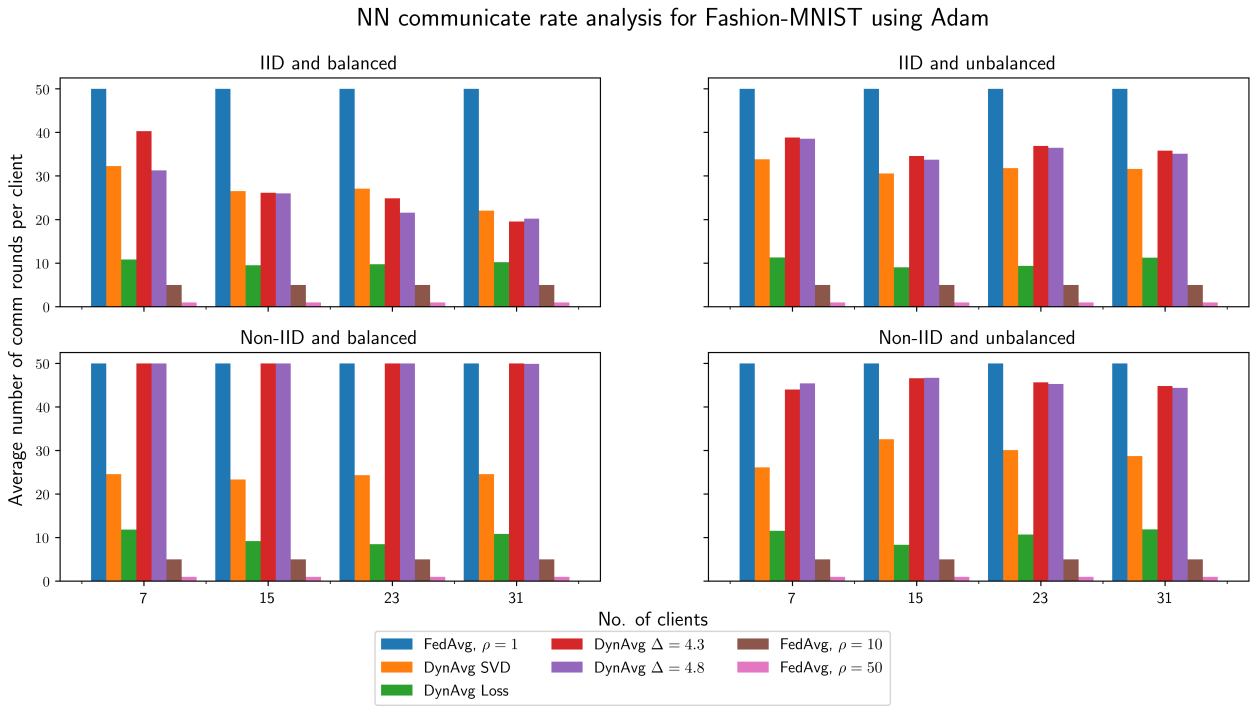

Figure 4.18: NN communication rate analysis across a different number of clients on Fashion-MNIST using Adam for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

us to provide entry level coverage into more complex models. We now consider a packet size analysis for CNNs. We refer to Figures 4.11, 4.15, 4.13, 4.17 for inspection.

MNIST and SGD. We briefly highlight that for CNNs we see similar results in terms of communication across all partitioning strategies as shown in Figure 4.19. Therefore, there is no need to explain in detail as the explanations remain similar to that of logistic regression and a simple neural network.

MNIST and Adam. All algorithms remain similar for MNIST using Adam besides DynAvg. The increase of communication for unbalanced experiments are once again evident for CNNS using Adam. The amount of communication becomes more pronounced for CNNs for DynAvg. This may further confirm that increasing the complexity of the model means adjusting the threshold or possibly the learning rate of Adam. We see a spike in communication across all partitioning strategies for DynAvg $\Delta \in \{2.2, 2.8\}$.

Fashion-MNIST and SGD. Figures 4.23 and 4.24 show there are no significant differences in communication metric trends when comparing Fashion-MNIST to MNIST for SGD for CNNs. We

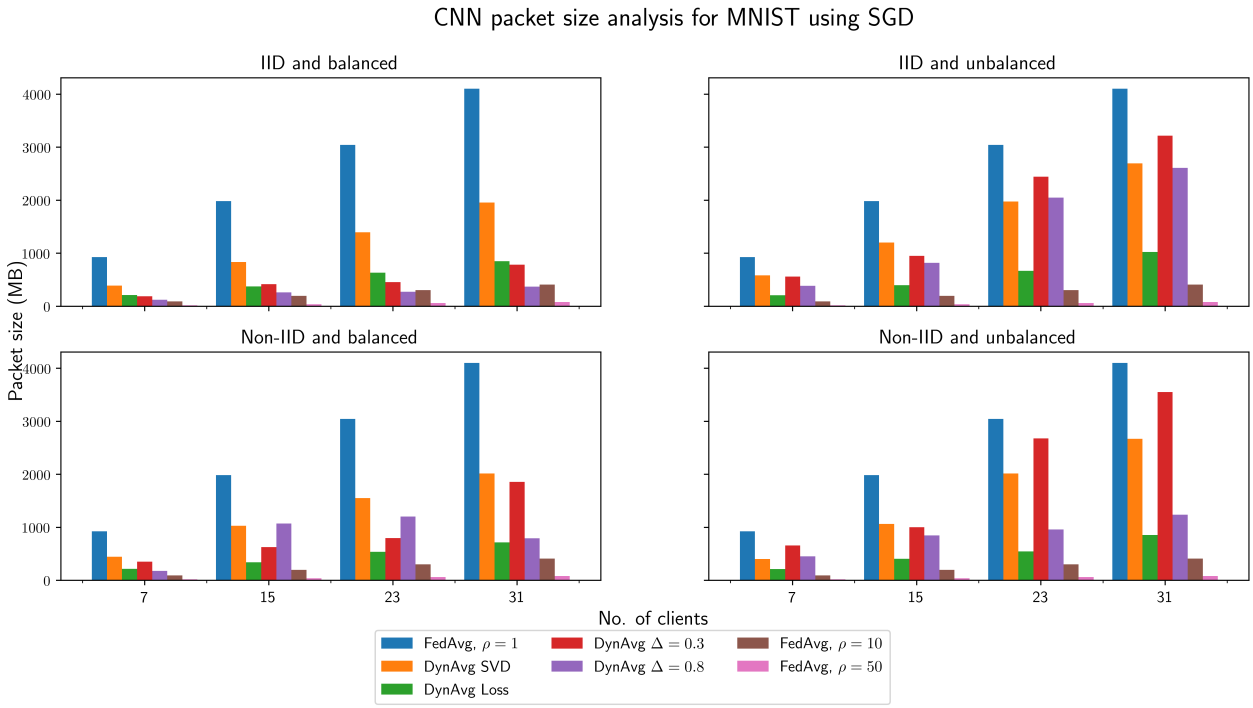

Figure 4.19: CNN packet size analysis across a different number of clients on MNIST using SGD for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

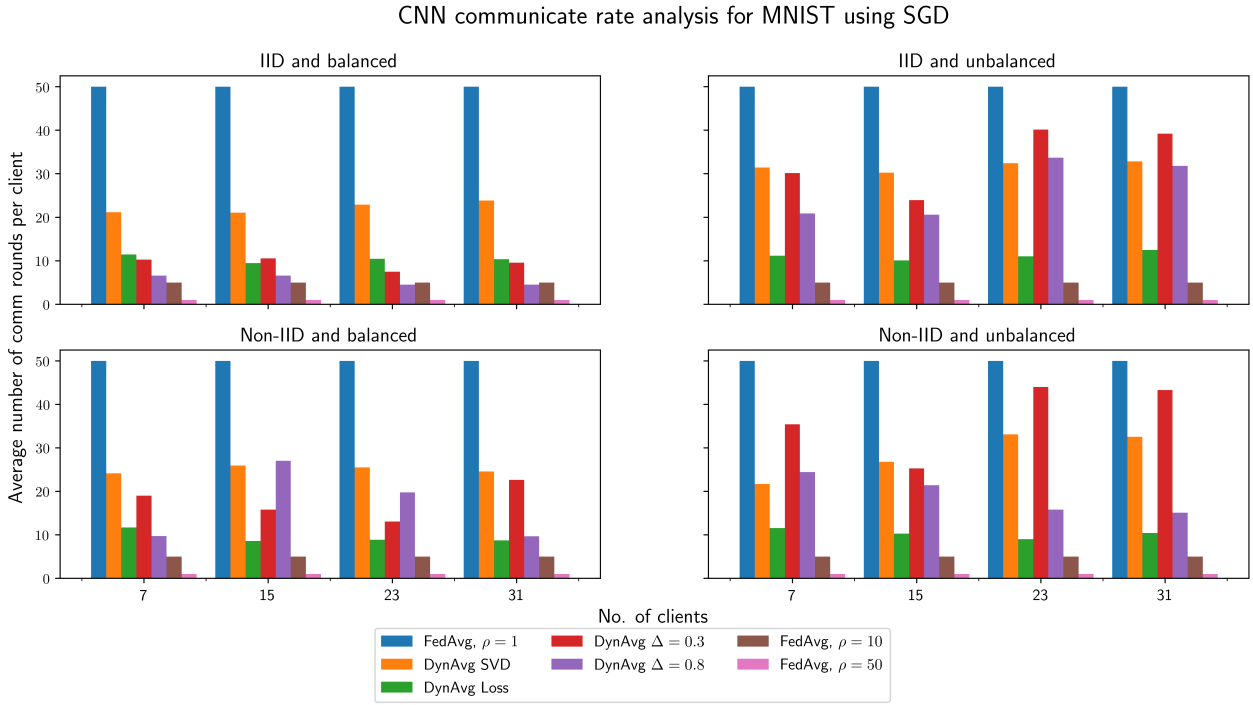

Figure 4.20: CNN communication rate analysis across a different number of clients on MNIST using SGD for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

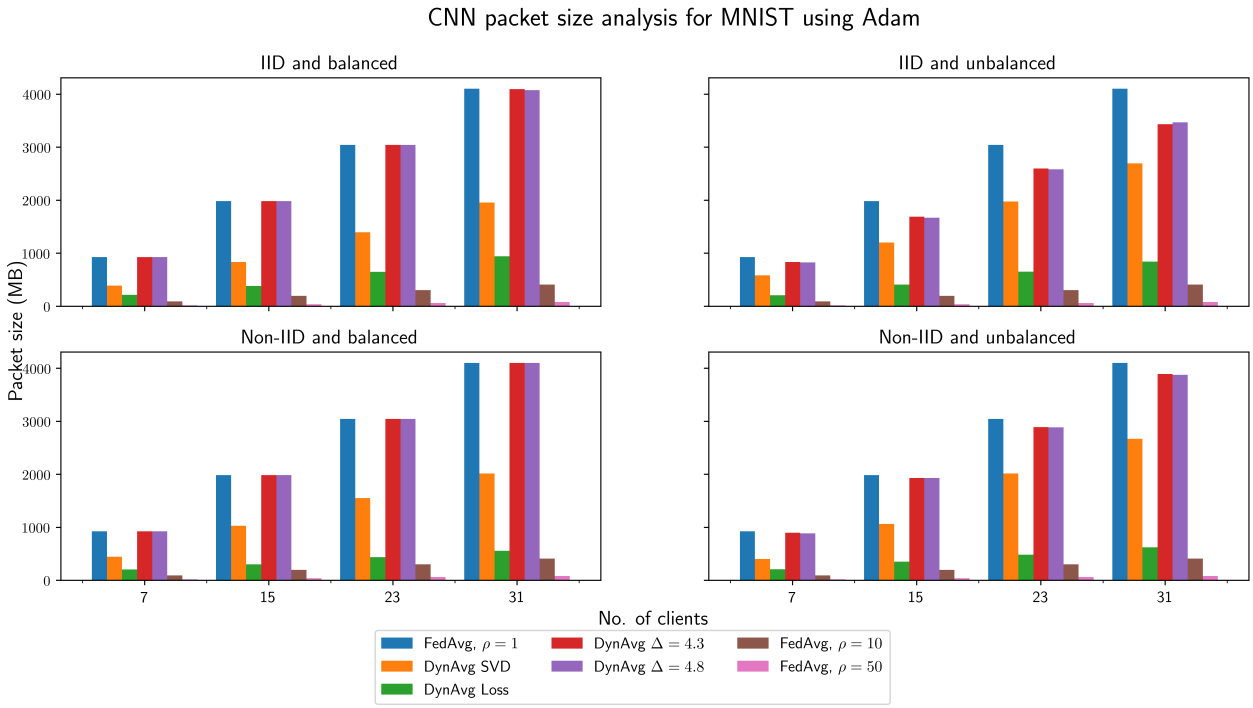

Figure 4.21: CNN packet size analysis across a different number of clients on MNIST using Adam for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

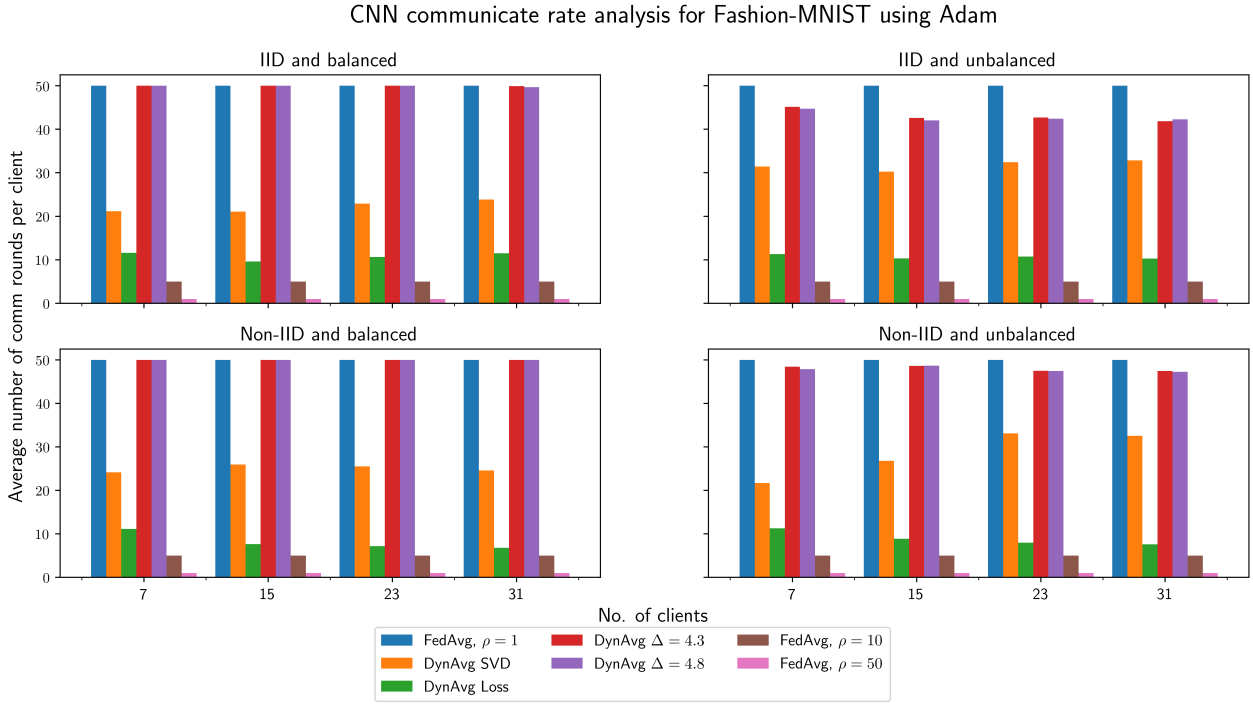

Figure 4.22: CNN communication rate analysis across a different number of clients on MNIST using Adam for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

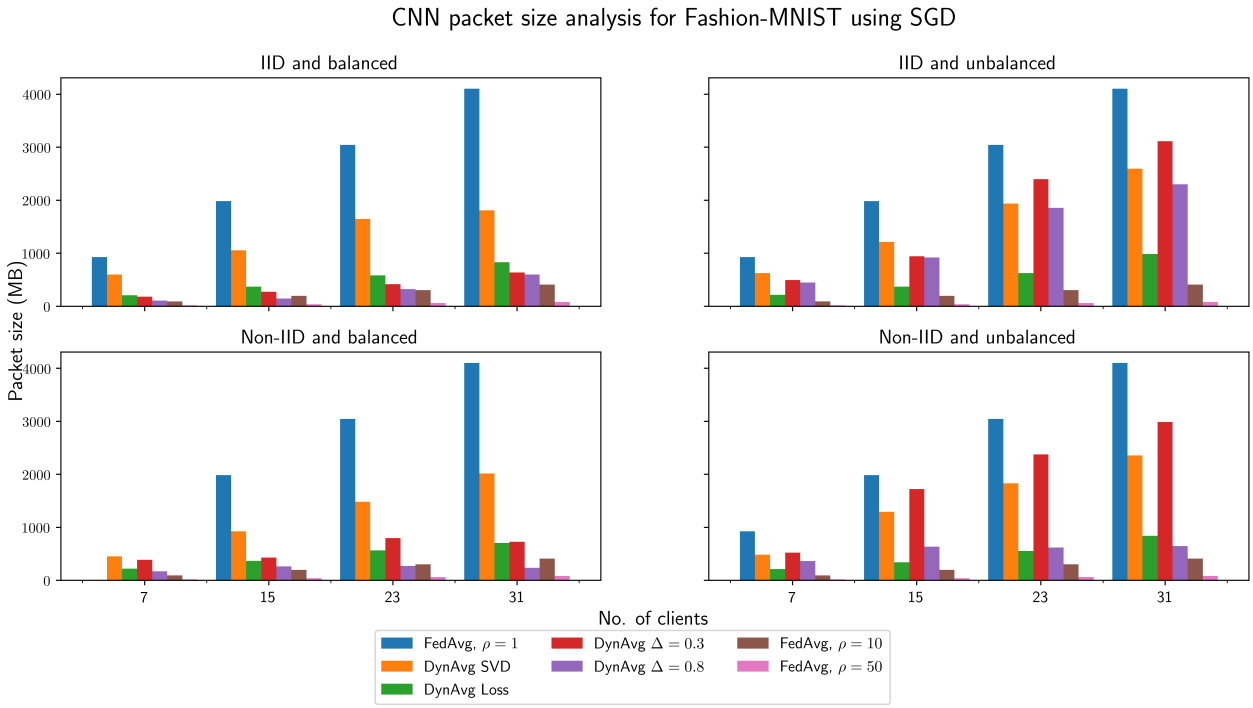

Figure 4.23: CNN packet size analysis across a different number of clients on Fashion-MNIST using SGD for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

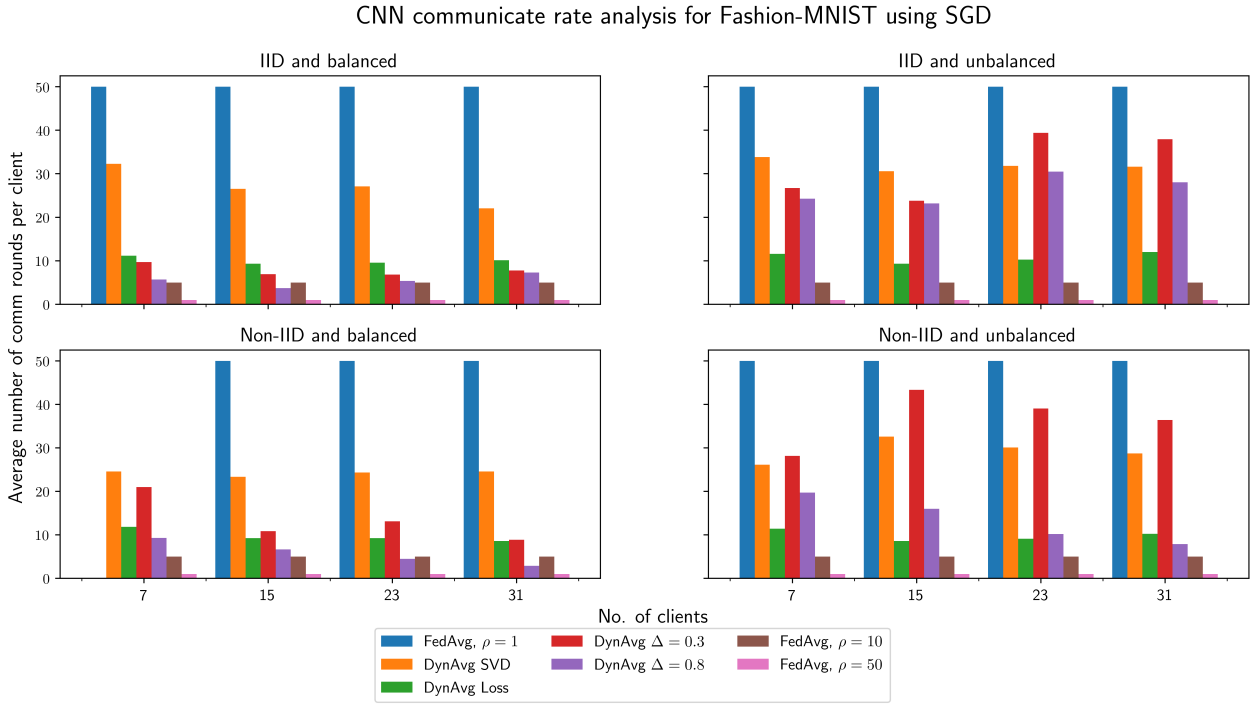

Figure 4.24: CNN communication rate analysis across a different number of clients on Fashion-MNIST using SGD for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

consistently see a linear increase across the number of clients for the different models.

We have consistently see an increase in communication for non-IID and balanced datasets thus far. This applies to CNNs as well. We have discussed this in detail in the previous sections for the previous models. The same applies in this section regarding CNNs.

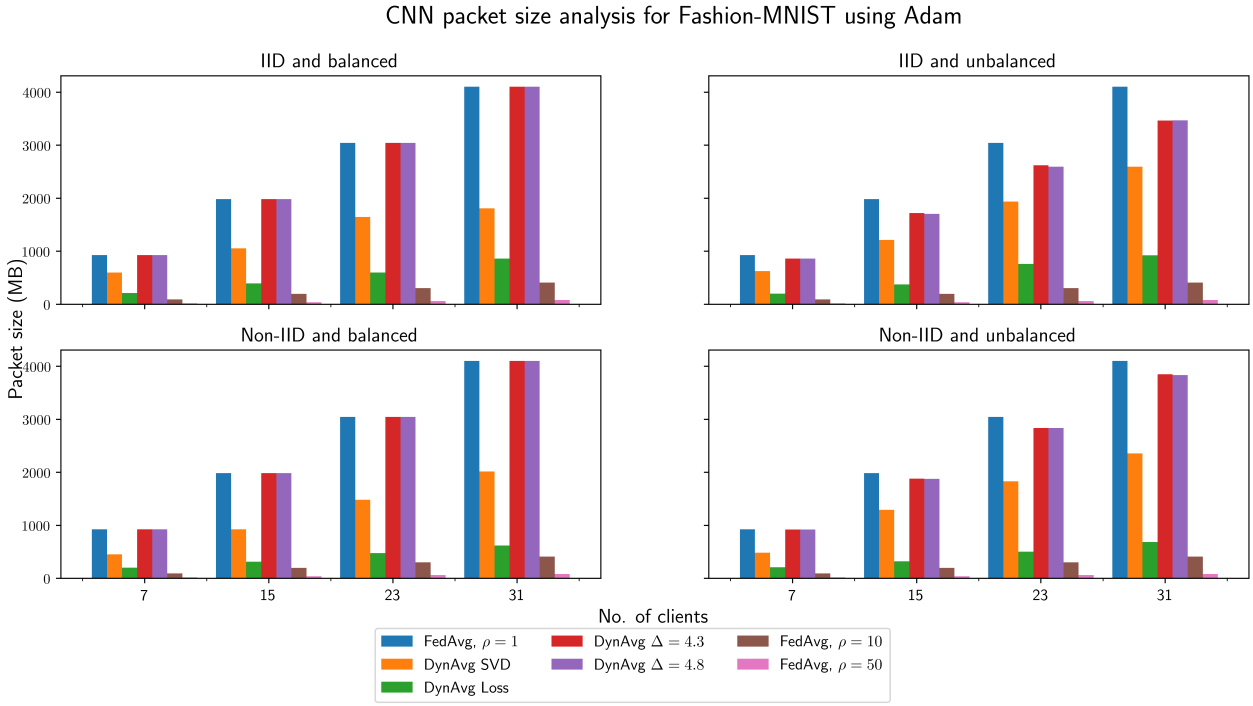

Figure 4.25: CNN packet size analysis across a different number of clients on Fashion-MNIST using Adam for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

Fashion-MNIST and Adam. It is evident again by Figures 4.25 and 4.26 that using Adam arises problems with investing communication efficiently. Similar to MNIST, this is more pronounced for CNNs using Adam on Fashion-MNIST. DynAvg once again is comparable to FederatedAveraging $\rho = 1$. The other algorithms remain similar in their communication strategies.

We have covered the communication rounds in details from the above figures. It is also important to consider the computation workload of the SVD computation. In order to do so, we briefly cover the sample distribution across the different number of clients. The sample distribution varies with the number of clients since we partition the data across the number of clients. This is shown by Figure 4.27. The number of samples vary due to the partitioning methods we describe in Section 3.1. In particular, the simulation of unbalanced datasets result in the maximum number of samples. For example, when there are 7 clients, the maximum number of samples for a client is equal to 22363 for

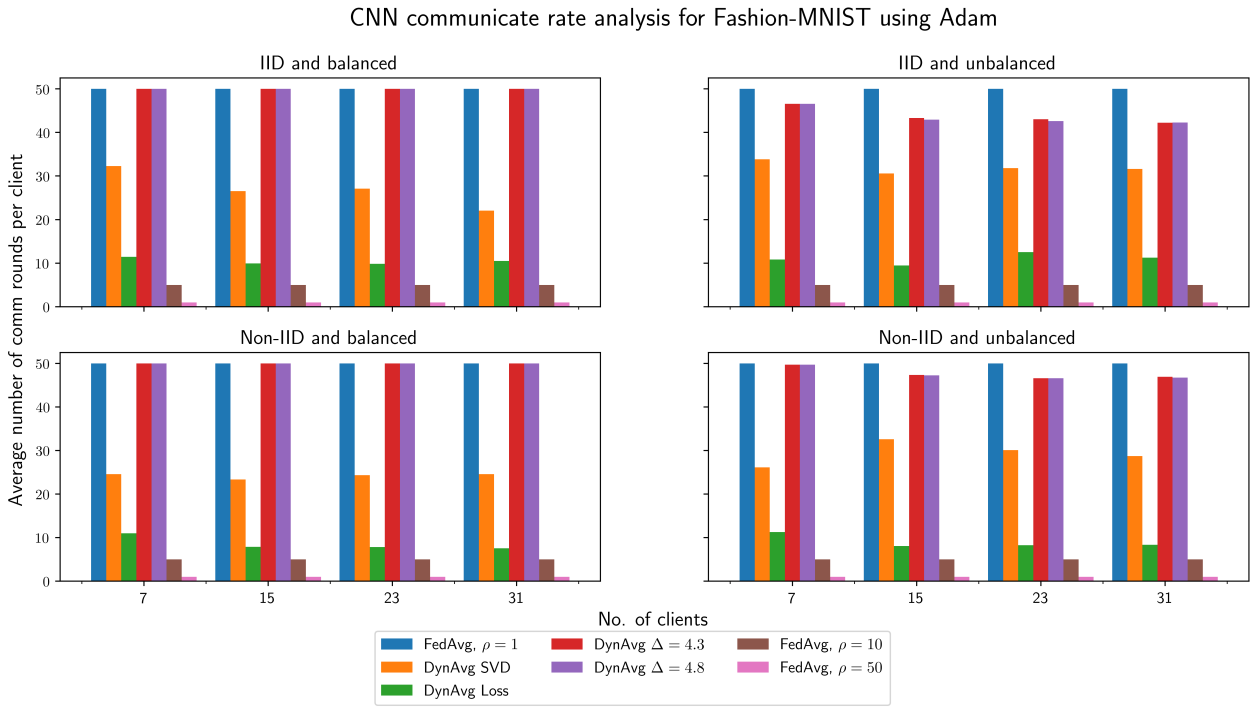

Figure 4.26: CNN communication rate analysis across a different number of clients on Fashion-MNIST using Adam for different communication protocols: FederatedAveraging, DynAvg, DynAvg SVD and DynAvg Loss.

MNIST and Fashion-MNIST.

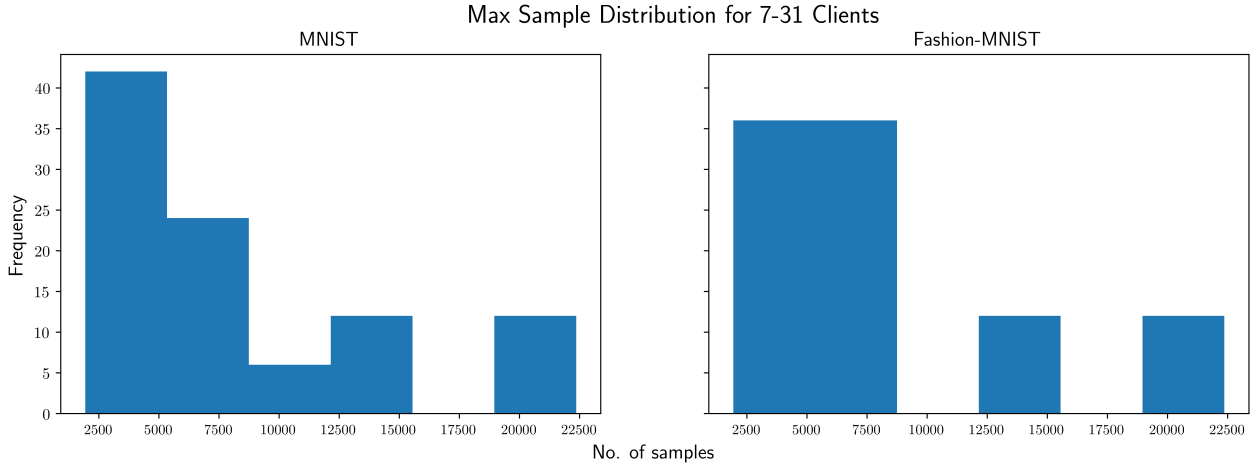

Figure 4.27: Max sample distribution across a different number of clients for MNIST and Fashion-MNIST for DynAvg SVD.

Figure 4.28 shows that the computation completes in 1.5-4.5 seconds on average for the MNIST dataset. The computation time is relatively consistent across the number of client's for the different partitioning methods. There is a slight increase in computation time when the number of clients

$K = 23$ for IID and balanced datasets. This is possibly due to the large variance when clients $K = 23$ shown by the large error bar. The large variance is possibly due to the inconsistency caused by the SLURM cluster.

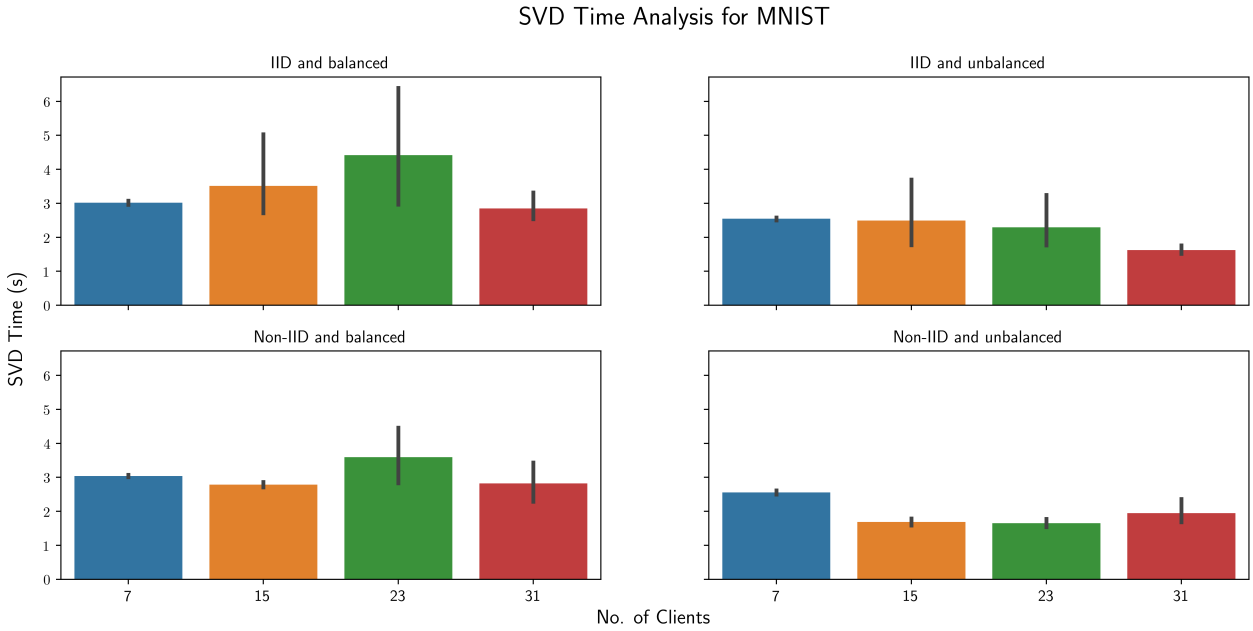

Figure 4.28: SVD Time analysis across a different number of clients on MNIST for DynAvg SVD.

The SVD time analysis for Fashion-MNIST provides a similar computation time range of 1.5-5 seconds. This is due to the total sample size of Fashion-MNIST being the same as MNIST. The inconsistency is evident when clients $K = 23$ for Fashion-MNIST similar to that of the MNIST dataset. We provide a similar explanation for the large variance.

We now look at model performance to get an indication if we invested the communication efficiently with negligible impact on model performance.

4.3 Effects of invested communication on model performance

We explore how the different communication protocols impact model performance in this section. We consider the accuracy of the model, more specifically the test accuracy. In addition, we provide insights into how well each different algorithm generalizes in a federated learning setting under different partitioning settings. To be clear and concise, we discuss up front how the results will be presented. We present results for each different dataset and optimizer which compare the different federated learning algorithms. Let's consider Figure 4.30 as an example. Along the left-hand y-axis we have accuracy. On the right-hand y-axis we have the generalization gap. These results are produced for a

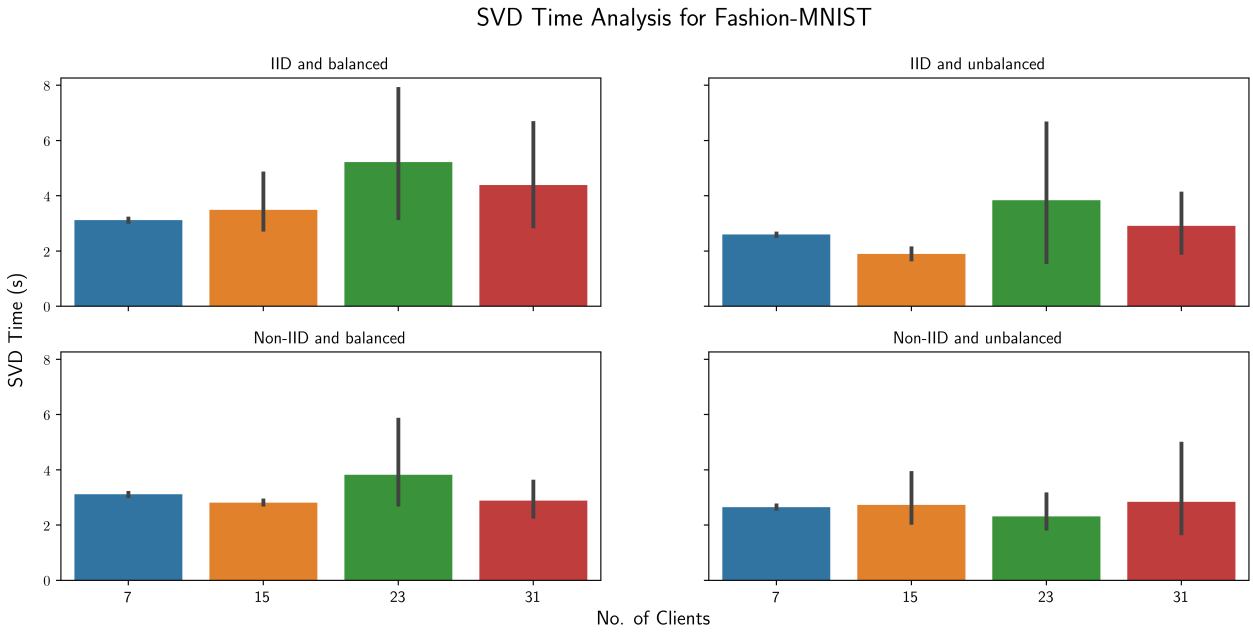

Figure 4.29: SVD Time analysis across a different number of clients on Fashion-MNIST for DynAvg SVD.

different number of clients which we display along the x-axis. The generalization gap G we refer to gives us an indication on how the model is overfitting which we discussed in Section 2.1. It can be calculated as per Eqn. (4.2)

$$G = \frac{J(\mathbf{w})_{test} - J(\mathbf{w})_{train}}{J(\mathbf{w})_{test} + J(\mathbf{w})_{train}}. \tag{4.2}$$

Eqn. (4.2) normalizes the difference between the train and test loss to be able to compare the result for different algorithms. If we obtain a positively large number, then this suggests there is signs of the model overfitting. If the difference is positively small or negative then the model is likely to not be overfitting.

Analyzing local datasets with SVD and influencing communication through local loss. We know from Section 3.3.1 that SVD allows us to analyze certain properties of a matrix. It is known how to calculate the dimension of the project SVD matrix which retains 95% of the variance for each client. Moreover, we know the properties of federated optimization and that the data is statistically heterogenous. The underlying question we want to also answer here as stipulated in Section 1 is as follows. Does SVD allow for efficient invested communication which has negligible impact on model performance in a federated learning setting?

Moreover, it was shown in Kamp et al. (2018) that using a per loss threshold retains loss bounds of periodically averaging protocols and is more efficient. Instead, this study uses a normalization technique to imitate the behaviour of Gradient Boosting (Friedman, 2000) to invest communication efficiently as explained in Section 3.3.2. We ask ourselves the same question for Algorithm 4. Does influencing communication through local loss have negligible impact on model performance in a federated learning setting? We look at the empirical evaluation broken down by the different types of models, datasets and optimizers. We first consider logistic regression.

4.3.1 Logistic regression

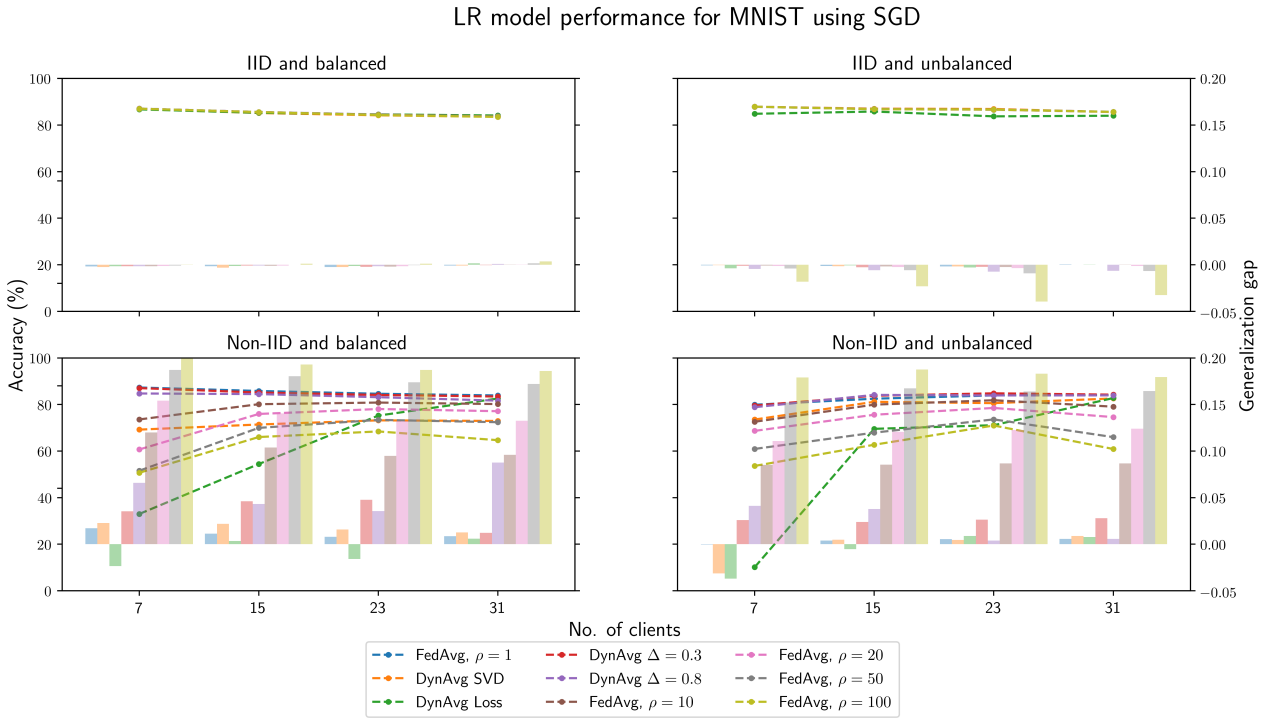

Figure 4.30: LR model performance for IID, non-IID, balanced and unbalanced data for MNIST using SGD.

MNIST and SGD. We know from our packet size analysis in Section 4.2 that we have improved communication efficiency in comparison to FederatedAveraging $\rho = 1$ for both algorithms 3 and 4. We observed a significant reduction in packet size (approximately 8x) in Section 4.2 for Algorithm 4 for dynamic averaging using loss as a mechanism to determine the communication protocol. For Algorithm 3 we observed approximately 2x reduction in accumulated packet size. Moreover, the accumulated packet size for Algorithm 3 is comparable to DynAvg $\Delta \in \{0.3, 0.8, 2.2, 2.8\}$.

Let us consider the IID case for both balanced and unbalanced cases in Figure 4.30 (upper plots). We observe that there is negligible difference in model performance between all experiments. Moreover,

we see minimal overfitting (along secondary y-axis) which is shown by the generalization gap. We also see that both Algorithm 3 and 4 hold well for IID cases for both balanced and unbalanced datasets as shown in Figure 4.30. There is little model degradation. This is expected since each client will have a good class distribution. Even when we consider unbalanced cases, the model performance does not degrade since the class labels are well distributed. Recall, that we are partitioning the data, so there will never be an overlap in the data samples. This may be an extreme case in the real world, but is suitable for federated learning. With the increase in the number of clients, the number of samples will decrease. This will impact performance since each local learner is observing less observations. Therefore, each client will not be able to generalize well. The gentle slope downwards is evidence that this does affect performance. We keep this in mind for all different experiments.

We do observe that there is model degradation for non-IID and balanced cases across all algorithms. There is an increase in the generalization gap for all algorithms. This is more so for FederatedAveraging $\rho > 1$. DynAvg is comparable to FederatedAveraging $\rho = 1$ despite communicating less which is shown by Figures 4.3 and 4.4. Even though Algorithm 3 does not show signs of overfitting, there is degradation in model performance. This is a common theme for non-IID cases for all models, datasets and optimizers. Non-IID cases are not a trivial problem to solve. The properties of non-IID data poses problems for all algorithms. Algorithm 3 relies on SVD to explore the underlying data distribution. However, if the data distributions do not assume similar probability distributions SVD struggles in these cases. All clients have at least two classes. For MNIST, that is two classes out of the possible ten. We know imbalanced datasets in the serial machine learning context pose significant problems. Non-IID is an extreme cases of traditional imbalanced datasets. Much like imbalanced datasets, it is evident that for non-IID cases additional measures need to be considered. This may require oversampling techniques, generative adversarial networks, data exchange methods (Verma et al., 2019) that use differential privacy.

We mentioned that FederatedAveraging with $\rho > 1$ is not designed to tackle non-IID and unbalanced cases. This comes through in Figure 4.30 for $\rho > 1$. Both Algorithms 3 and DynAvg perform better than FederatedAveraging $\rho > 1$. In fact, Algorithm 3 is comparable with the state-of-the-art DynAvg $\Delta \in \{0.3, 0.8\}$ technique. We recognize that Algorithm 4 shows some instability across the number of clients. Recall that Figure 4.3 we observed that Algorithm 3 and Algorithm 4 communicate less than DynAvg methods for non-IID and unbalanced datasets. A common theme we observe for Algorithm 3 is that it does not overfit. However, in non-IID and balanced cases, it does underfit. This is less so for

non-IID and unbalanced cases as it is comparable with the state-of-the-art techniques. The benefits of using SVD come through for unbalanced datasets. This is since clients will have notably different data distributions and hence invest it's communication efficiently. We also note that it does not require any hyperparameter tuning to determine the communication protocol. When we compare this to DynAvg, we see that we have to carefully consider the threshold Δ. We notice how this impacts communication if set incorrectly in Section 4.2. Algorithm 4 suffers quite a bit of model degradation. This may be since the communication protocol is too dynamic. We constantly change the communication interval for each client every iteration. This may result in switching focus too often on which clients should contribute more to training. Thus, resulting in searching too wide of a space and not converging. This could possibly be improved upon, by only calculating different communication intervals for a small number of epochs. Thereafter, the interval will be kept constant. This is something we consider for future work.

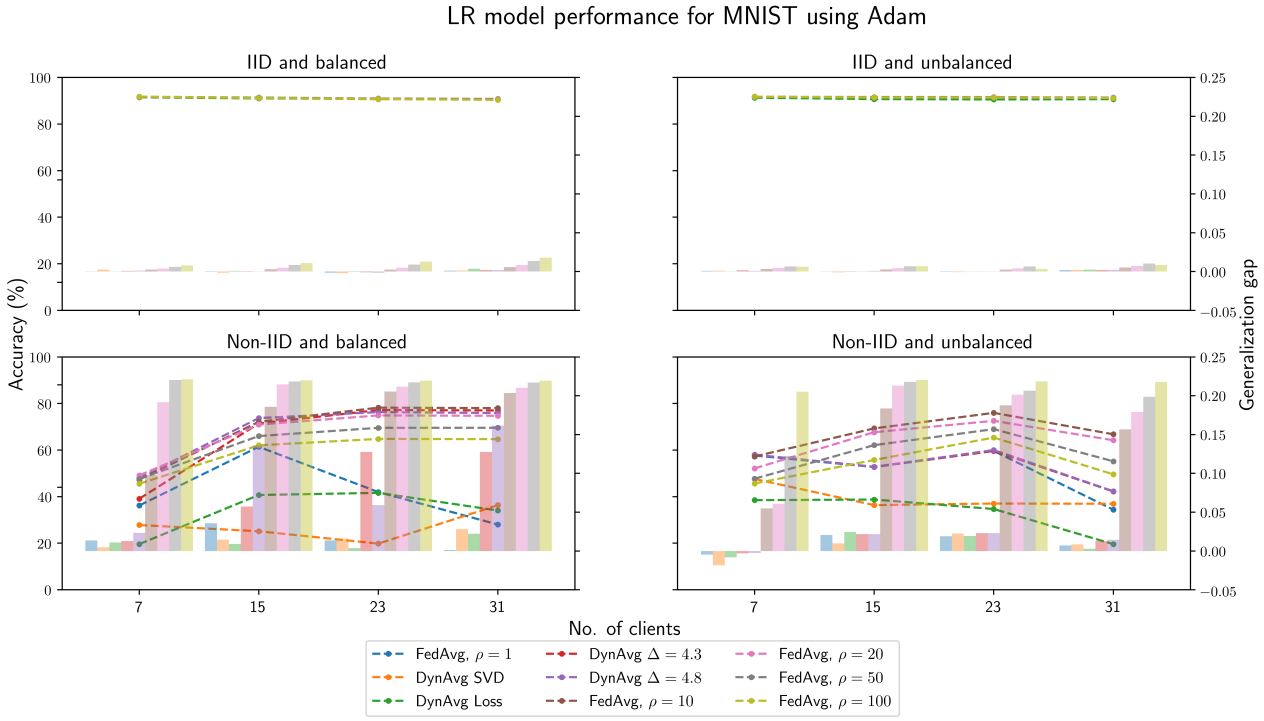

Figure 4.31: LR model performance for IID, non-IID, balanced and unbalanced data for MNIST using Adam.

MNIST and Adam. We see a similar trend for MNIST using Adam for logistic regression in comparison with using SGD shown by Figure 4.31. There is negligible impact on performance for both IID cases for all algorithms. We do highlight that for Adam, the thresholds selected meant DynAvg communicates more often. Multiple different thresholds were set in the range of $[0.3, 4.8]$. Lower

thresholds mean that violations are more regular for Adam. There is a slight reduction in accumulated packet size as this threshold increases. The thresholds used by the authors of DynAvg (Kamp et al., 2018) do not highlight any details about how to pick a divergence threshold for different optimizers. The communication metrics are comparable with FederatedAveraging $\rho = 1$. Figures 4.6 and 4.31 show that for IID cases we are able to perform one shot averaging for logistic regression without model degradation. This is for both Adam and SGD.

For non-IID and balanced cases, we once again observe model degradation across all algorithms. This highlights that we need to employ some techniques to deal with the significantly class distributions at each client. Dynamic averaging techniques suffer slightly. Clients that have a higher loss should not necessarily communicate more often in this setting as per Algorithm 4. This is due to the different distributions across clients. The same goes for Algorithm 3 but from a SVD perspective. FederatedAveraging does not assume anything about the model or the data distribution. We also see the flaw in this.

The trend continues with non-IID and unbalanced cases. Model degradation is evident across the number of clients for Adam. More so than SGD in Figure 4.30.

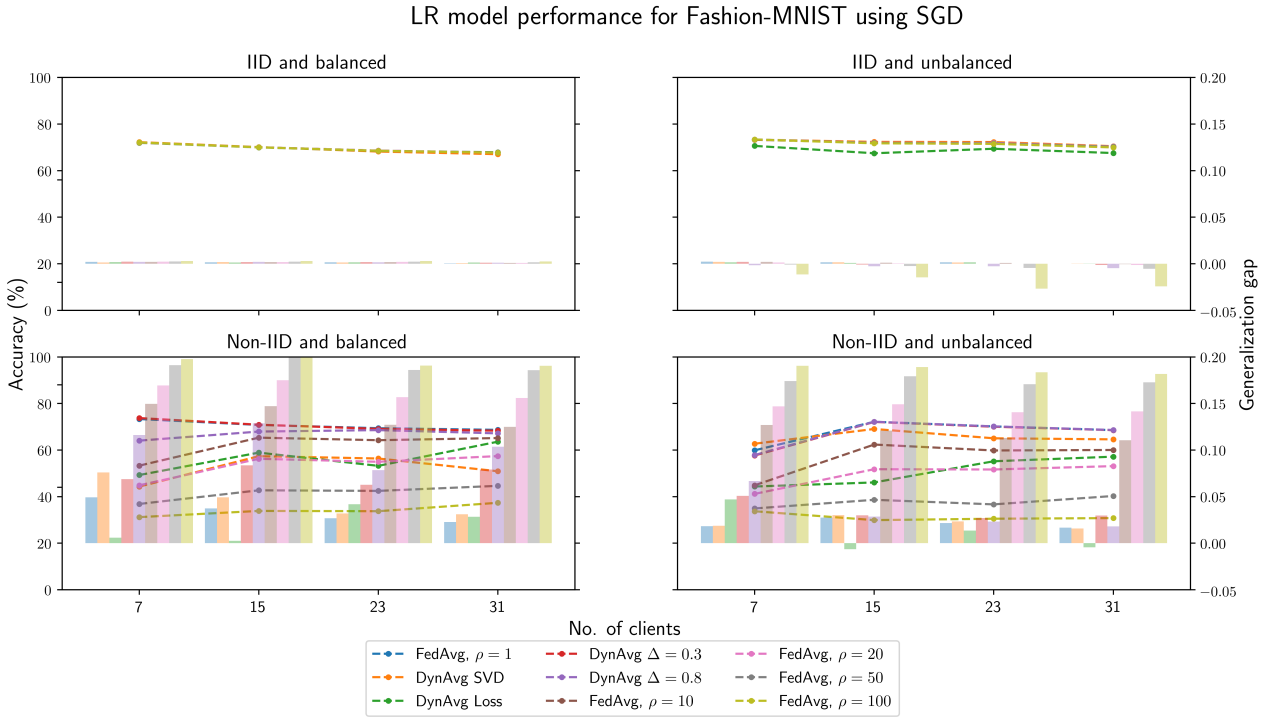

Figure 4.32: LR model performance for IID, non-IID, balanced and unbalanced data for Fashion-MNIST using SGD.

Fashion-MNIST and SGD. We skip over the IID cases for both unbalanced and balanced cases. There is minimal model degradation across the different algorithms. Moreover, there is minimal overfitting. If we achieve a lower accuracy for an algorithm when comparing it to other algorithms, then relatively speaking we are underfitting.

Results for Fashion-MNIST using SGD yield similar trends to MNIST for non-IID and balanced cases. Not necessarily the shape of the graphs, but the fact that we see model degradation. This make sense, as the partitioning strategies are the same across datasets.

For non-IID and balanced cases we see that Algorithm 3 is comparable to DynAvg $\Delta = 0.3$ and FederatedAveraging $\rho = 1$ in terms of model performance. The also have a similar generalization gap. Despite Algorithm 4 not overfitting, it appears to be underfitting as we see average model performance relative to all other algorithms.

Figure 4.33: LR model performance for IID, non-IID, balanced and unbalanced data for Fashion-MNIST using Adam.

Fashion-MNIST and Adam. There appears to be significant model degradation for algorithms on Fashion-MNIST using Adam shown by Figure 4.31. For IID cases, we see an improvement in model accuracy as one would expect for learners that use Adam. However, for non-IID cases, we see a drop in

model accuracy across all algorithms. FederatedAveraging $\rho = 1$ which communicates every iteration also suffers similar issues. A possible reason may be that each local learner is learning too fast and diverging at a rapid rate with Adam. Even though there is a synchronization step, the search space may be too different since the data distribution is vastly different.

4.3.2 Simple neural network

We transition from a convex objective function in logistic regression to a non-convex objective function of neural networks. We aim to answer whether dynamic averaging is suitable in non-convex settings. We utilize the same settings for a simple neural network as we did for logistic regression.

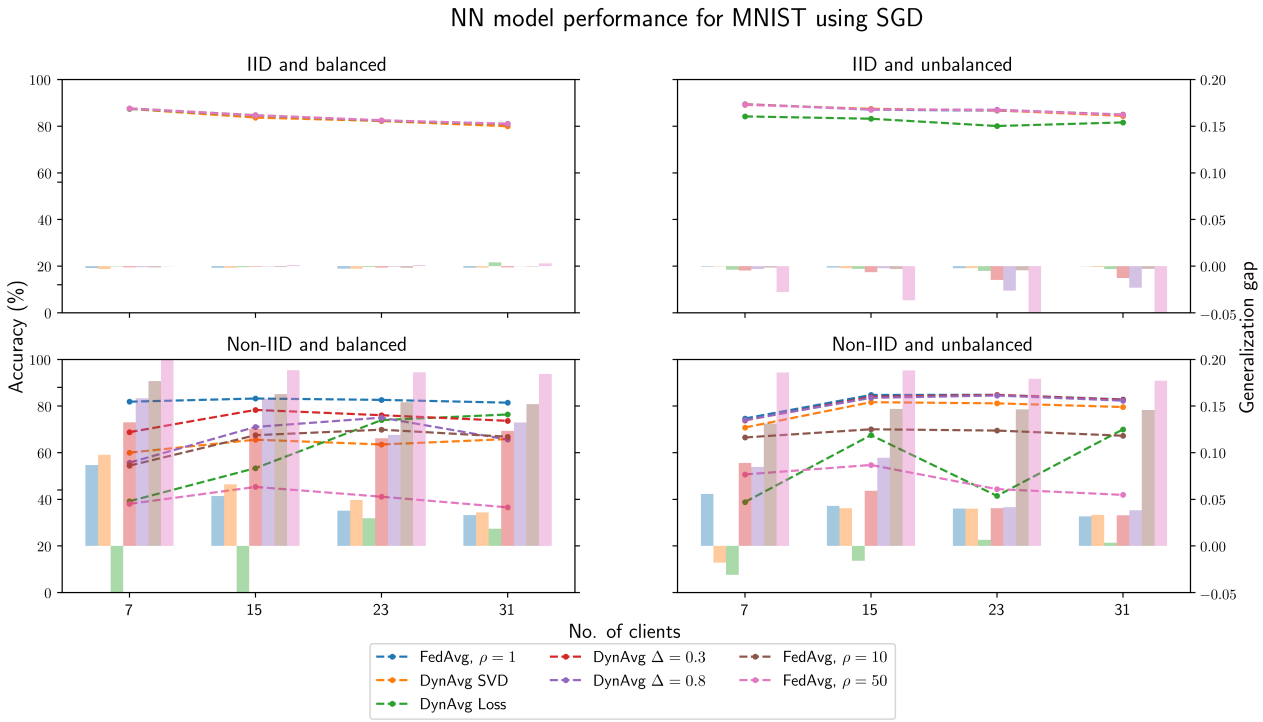

Figure 4.34: NN model performance for IID, non-IID, balanced and unbalanced data for MNIST using SGD.

MNIST and SGD. We consider MNIST with SGD as our optimizer in this section. Figure 4.34 shows that all algorithms perform well for IID cases in a non-convex setting. This is similar to what we encountered in a convex setting.

Figure 4.34 also suggests that model degradation in non-convex settings for non-IID and balanced cases is similar to that of a convex setting. We observe similar model degradation that we observed in Figure 4.30. For FederatedAveraging, model accuracy decreases as ρ increases. Moreover, the generalization gap increases as ρ increases. We have mentioned previously that more synchronizations

lead to lower loss or error rates. Algorithm 3 suffers in a non-convex setting under the non-IID and balanced dataset setting. With non-convex settings, we introduce multiple local minima. Despite this, gradient based methods perform well in practice. However, gradient based methods and machine learning algorithms are dependent on the data distributions. We have discussed in great detail the issues of non-IID and balanced cases. This is also prevalent in experiments utilizing a simple neural network.

Figure 4.34 shows Algorithm 3 being comparable to FederatedAveraging $\rho = 1$ and DynAvg $\Delta \in \{0.3, 0.8\}$. However, from Figure 4.12, we noticed an increase in communication for DynAvg. Expectedly, more synchronizations lead to better model performance. Currently, this tradeoff would be left up to the user developing the model in a federated learning setting. Future work will include handling this problem by devising a set of guidelines when dealing with this tradeoff. Algorithm 4 seems to show instability under this partitioning strategy once again. In a non-convex setting it seems to suffer from similar problems as it did in the convex setting. We have discussed possible reasons for this and improvements when we considered logistic regression. These improvements are applicable for neural networks as well.

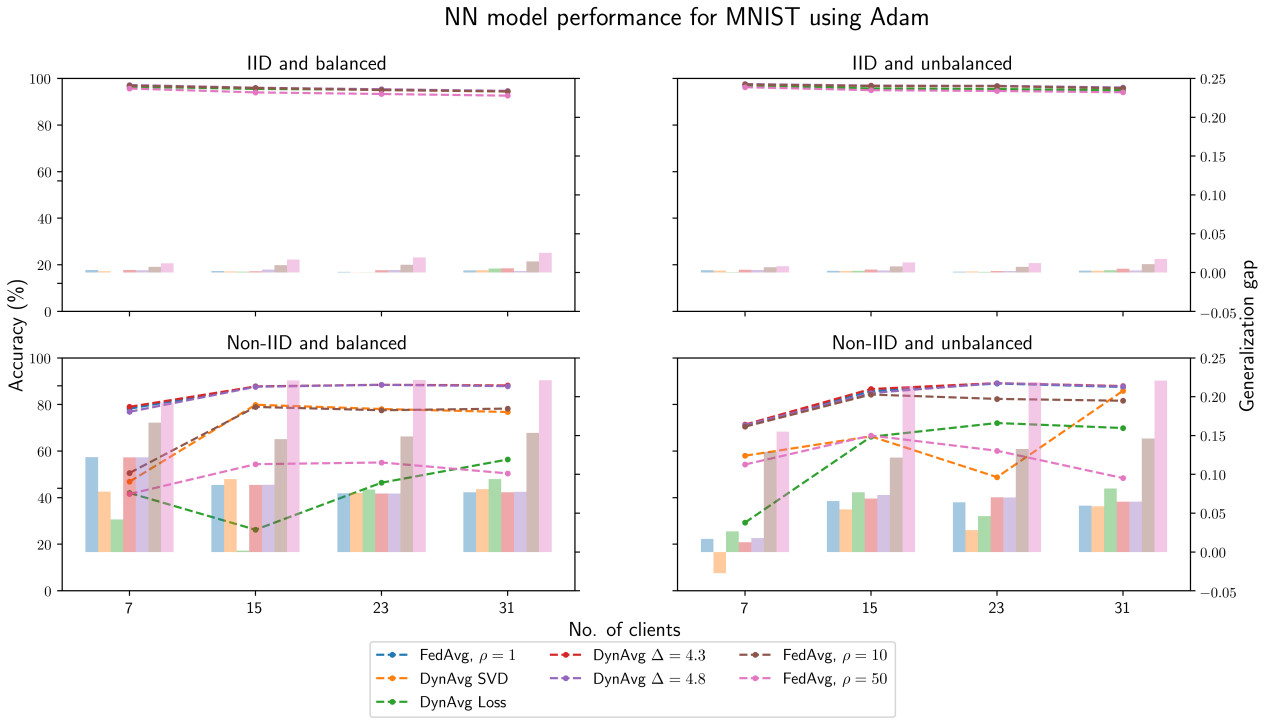

Figure 4.35: NN model performance for IID, non-IID, balanced and unbalanced data for MNIST using Adam.

MNIST and Adam. We have seen in Section 4.2.1 that using Adam under the different partitioning settings presents some interesting results. For both IID cases, the results remain similar to that of logistic regression that we showed in Section 4.2.1.

We have discussed the issues that non-IID datasets bring. Expectedly, there is model degradation in non-IID and balanced cases for all algorithms. This is less so for FederatedAveraging $\rho = 1$ but there is still decrease in performance if we compare to the IID cases. Both Algorithms 3 and 4 suffer from a decrease in performance. It is hard to comment on DynAvg techniques Figures 4.13 and 4.14 show that they communicate almost as much as FederatedAveraging $\rho = 1$.

As we observed for logistic regression using Adam, we see FederatedAveraging $\rho = 1$ perform the best. DynAvg is comparable to Algorithm 3. Algorithm 4 suffers from similar model degradation to FederatedAveraging $\rho > 1$. We can assume the same explanations apply for neural networks as logistic regression experiments for MNIST using Adam as the optimizer.

Figure 4.36: NN model performance for IID, non-IID, balanced and unbalanced data for Fashion-MNIST using SGD.

Fashion-MNIST and SGD. Figure 4.36 shows that we can once again skip discussions for IID cases. There is negligible performance for both balanced and unbalanced datasets. We note that model

performance for a neural network is comparable to logistic regression for Fashion-MNIST. However, we have not fine-tuned the neural network to achieve optimal results i.e. adjust the learning rate, etc. This does not pose as a problem as we want to get a clear indication of how the different algorithms compare with one another in different model spaces. Similarly to logistic regression for Fashion-MNIST, there is model degradation in a non-convex setting. This is a common theme for neural networks. For non-IID and unbalanced datasets, FederatedAveraging $\rho > 1$ decreases in the model accuracy and increases the generalization gap as ρ increases. DynAvg is comparable to FederatedAveraging $\rho = 1$ but does communicate more often than Algorithm 3. We observe that Algorithm 4 once again does not hold for non-IID and unbalanced datasets.

Figure 4.37: NN model performance for IID, non-IID, balanced and unbalanced data for Fashion-MNIST using Adam.

Fashion-MNIST and Adam. The trends we observe using Adam for Fashion-MNIST is similar to the trends we observed when using SGD as an optimizer. However, Algorithm 3 performs slightly worse for both non-IID cases. In Figure 4.33 we saw this similar trend for logistic regression whereby all models suffered model degradation more so than SGD. Our assumption is similar, in that we assume that a faster learner on data that is statistically heterogenous causes enhanced divergence.

4.3.3 Simple convolutional neural network

We progress from a simple neural network in the non-convex setting to a more complex model. We present results for a simple convolutional neural network. The architecture for this CNN is simple when compared to the state-of-the-art CNN architectures but provides entry level coverage of more complex models. We aim to answer the same questions we posed for the other models. Moreover, we wish to see how adding more complexity to the model in a federated learning setting impacts model performance. We have already discussed communication metrics in Section 4.2.3 for CNNs. The trends are similar to logistic regression and NNs in terms of communication metric trends. So far, we have also seen that logistic regression and neural network model performance trends have shown some similarities. We have also seen how Adam differs from SGD under different the different partitioning strategies, particularly for non-IID cases. In the forthcoming section, we look at how CNNs perform for different experiments.

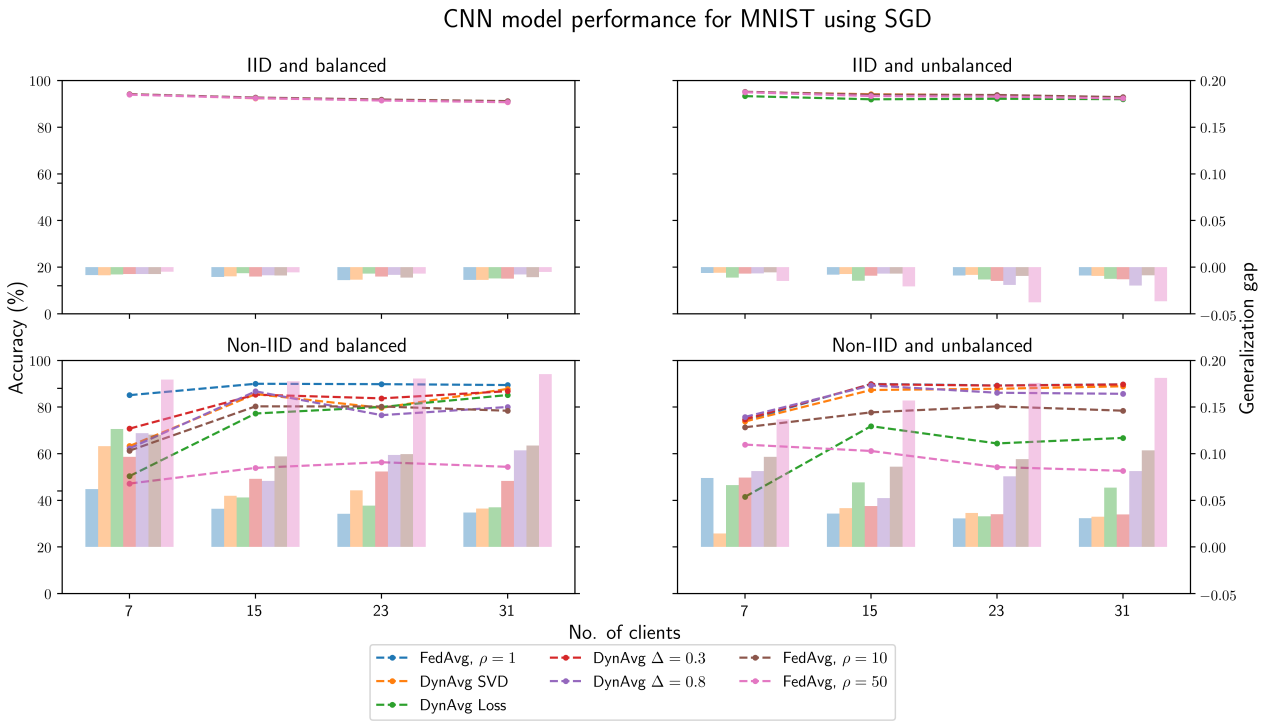

Figure 4.38: CNN model performance for IID, non-IID, balanced and unbalanced data for MNIST using SGD.

MNIST and SGD. We skip straight to the non-IID cases for CNNs, since we do not observe any differences for IID cases. Figure 4.38 shows that all algorithms are expectedly similar due to the data being identical and independently distributed. This property seems to be most key in a

federated learning setting if we consider the model performance results thus far. We do observe slightly model degradation for all algorithms. However, we see that all algorithms are comparable excluding FederatedAveraging $\rho = 50$. This is one-shot averaging, so we expect that model performance be the worst for this model setup. It can be observed that Algorithm 3 is comparable to DynAvg $\Delta \in \{0.3, 0.8\}$. Algorithm 4 is also comparable to DynAvg and Algorithm 3 but for clients $K \in \{15, 31\}$. It appears that for less clients there is a reduction in model performance across all algorithms.

Figure 4.38 shows that Algorithm 3 is once again comparable to DynAvg and FederatedAveraging $\rho = 1$. We recall that Figures 4.20 and 4.19 showed a reduction in communication compared to DynAvg. Moreover, there was no need to set parameters which may impact the communication protocol.

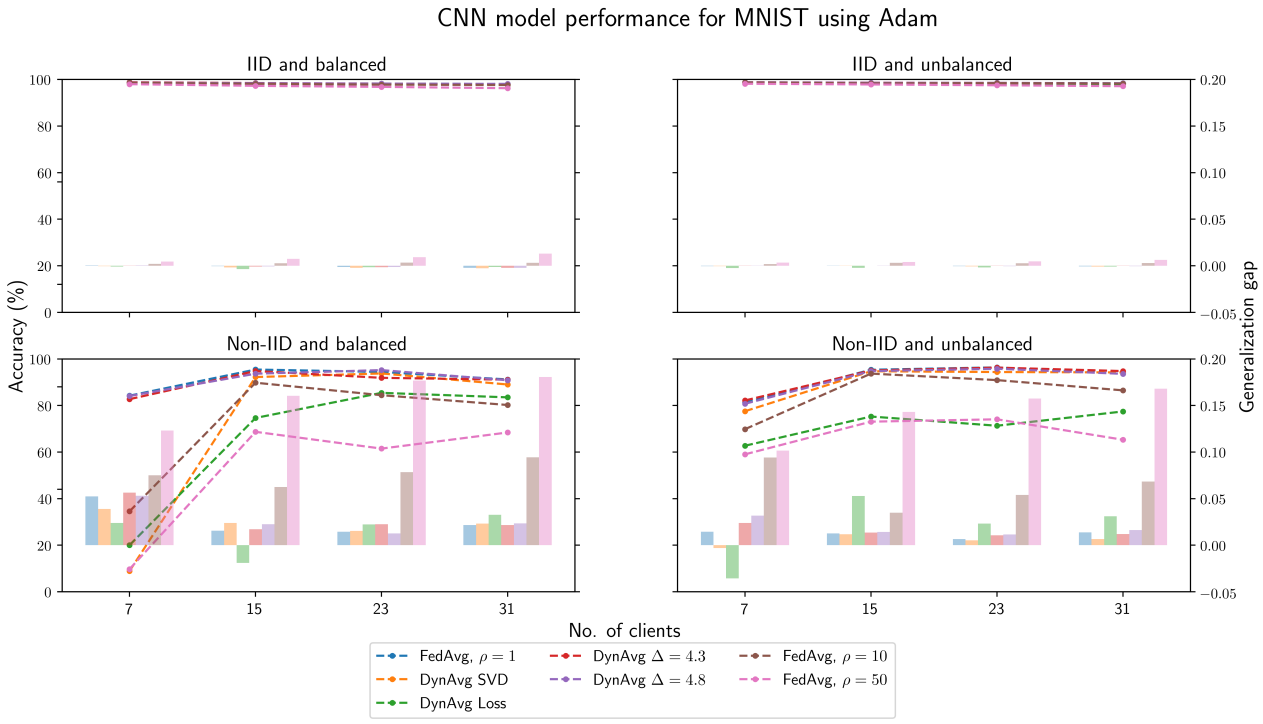

Figure 4.39: CNN model performance for IID, non-IID, balanced and unbalanced data for MNIST using Adam.

MNIST and Adam. For CNNs, we observe that there is less model degradation across all partitioning strategies than for a simple neural network which we showed in Section 4.3.2. For non-IID and balanced cases we see that Algorithm 3 is comparable to FederatedAveraging $\rho = 1$ and DynAvg respectively. Note that a lot more communication is invested to achieve those results for both FederatedAveraging $\rho = 1$ and DynAvg. This is shown by Figure 4.19. In this case, we see the benefit of determining the communication protocol based on SVD. This also goes for the non-IID and unbalanced case. We see a similar trend whereby these state-of-the-art techniques invest a lot more communication

in their communication protocols. Despite this, Algorithm 3 is comparable with these algorithms. FederatedAveraging $\rho > 1$ suffers in the non-IID cases as it did for the other two types of models in Sections 4.3.1 and 4.3.2. Algorithm 4 suffers in the non-IID cases and does not perform well. We have discussed the downfalls of this algorithm. Those discussions also hold for CNNs.

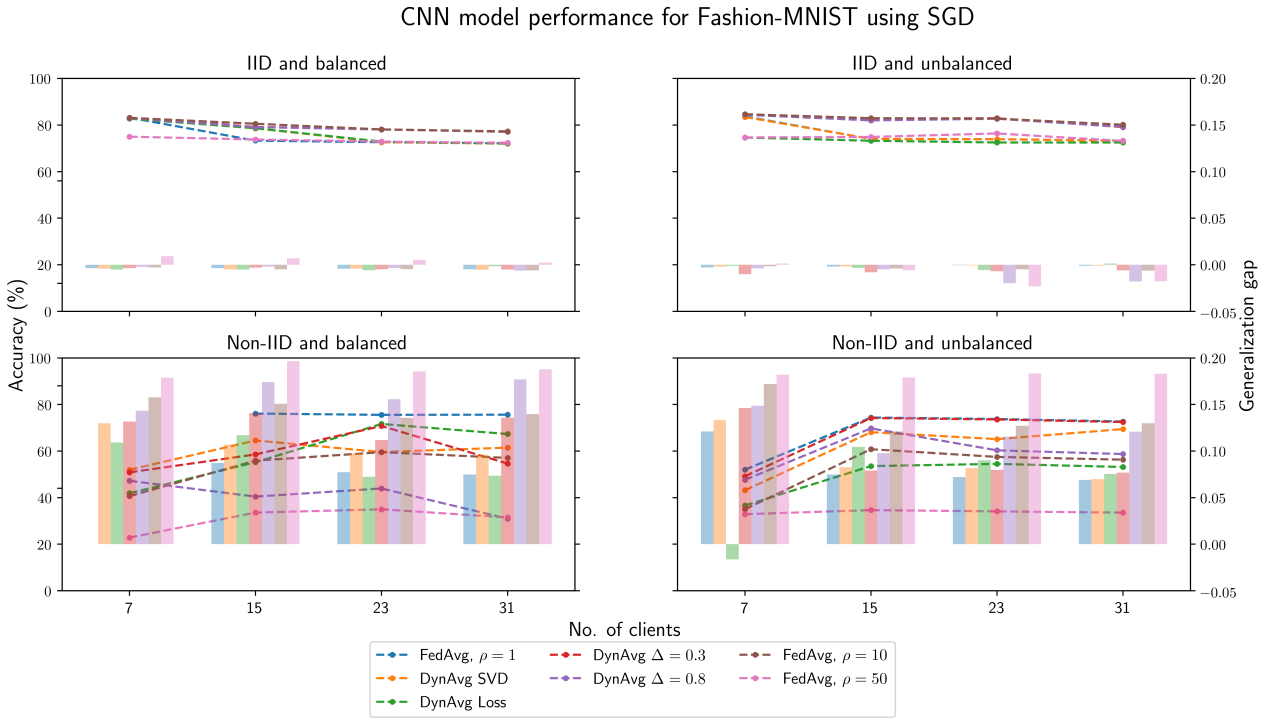

Figure 4.40: CNN model performance for IID, non-IID, balanced and unbalanced data for Fashion-MNIST using SGD.

Fashion-MNIST and SGD. Let us consider the Fashion-MNIST dataset using SGD for CNNs. Figure 4.40 shows that for IID and balanced cases, there is minimal model degradation. However, algorithms that synchronize less often appear to suffer slight model degradation such as FederatedAveraging $\rho = 50$. It is possible that less synchronization for more complex model has an effect on model performance. This is a similar trend for IID and unbalanced cases. We observe that algorithms that communicate more often such as FederatedAveraging $\rho = 1$, suffer no model degradation.

We observe the similar trend of model degradation for non-IID cases for both balanced and unbalanced datasets. FederatedAveraging $\rho = 1$ also suffers slight model degradtion. This highlights the problem that non-IID datasets present. Algorithms 3 and 4 both suffer a decrease in model performance but are comparable with DynAvg $\Delta = 0.3$. For unbalanced datasets shown by the bottom right plot in Figure 4.38, we observe that Algorithm 3 is most comparable to DynAvg $\Delta = 0.3$. This being without

having to set hyperparameters to achieve similar results. We notice that when setting $\Delta = 0.8$, the model performance degrades. Algorithm 4 suffers in terms of model performance for non-IID and unbalanced datasets. We regularly see that it is comparable to FederatedAveraging $\rho > 1$ in terms of model performance. If we consider the communication metrics we observed in Figures 4.23 and 4.24. Algorithm 4 is similar to that of FederatedAveraging $\rho > 1$. However, the instability of Algorithm 4 shows that even though we communicate efficiently, we do not invest communication well.

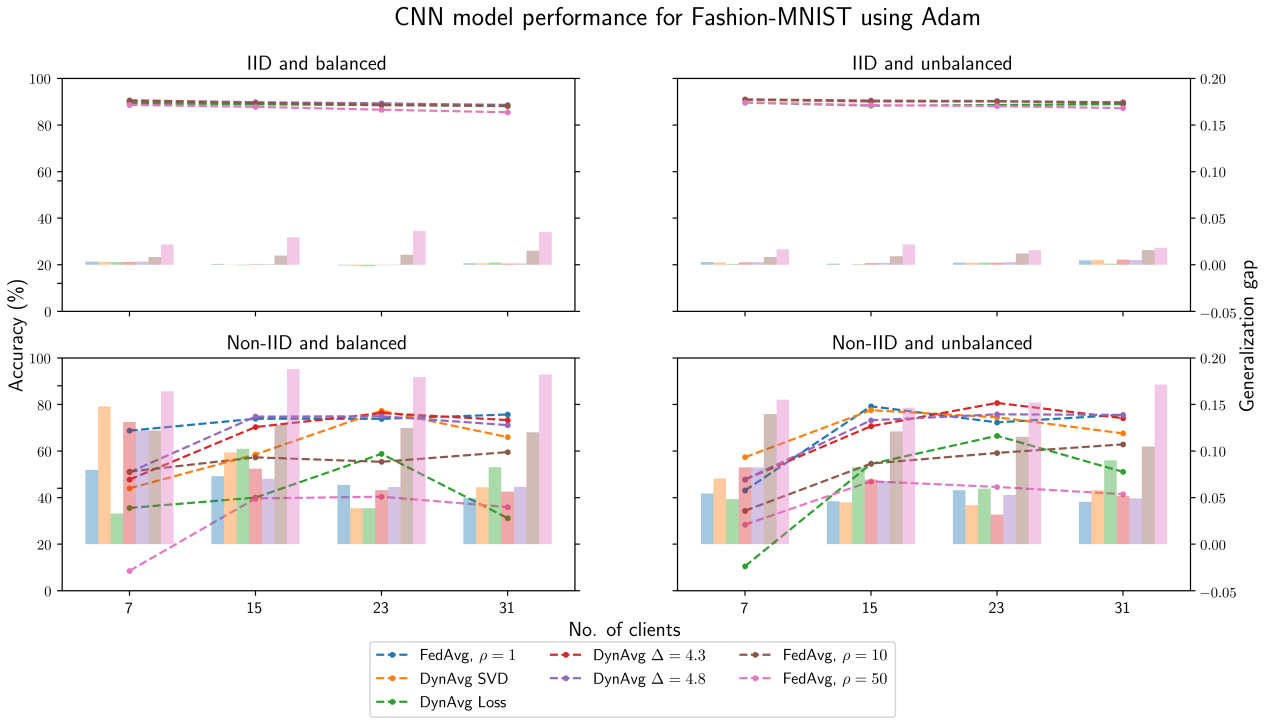

Figure 4.41: CNN model performance for IID, non-IID, balanced and unbalanced data for Fashion-MNIST using Adam.

Fashion-MNIST and Adam. Using Adam for Fashion-MNIST on CNN models do not present any unexpected results. Figure 4.41. For both IID cases, we expectedly observe minimal model degradation. This observation is for all the algorithms that were used to run these experiments.

By now, we have come to expect model degradation for non-IID cases for all the algorithms. We emphasize that the algorithms presented in this study need to be combined with oversampling techniques or data exchange methods. That being said, we observe that Algorithm 3 is comparable to DynAvg for non-IID cases. Also note by Figures 4.25 and 4.26, we observed that DynAvg methods endured a greater communication cost. We observe similar trends for Algorithm 4 for CNNs as we did for NNs and logistic regression.

In summary. Throughout this section, we observe a common theme amongst the different model types. Solving for minimal impact on model performance for non-IID cases is not trivial. It also highlights that successful federated learning algorithms need to focus on the key properties of federated learning which we discussed in Section 2.4. Most algorithms solve for unbalanced datasets when they are identically and independently distributed. However, for non-IID cases, oversampling techniques or data exchange methods are required.

We can take away that there is direct correlation with the underlying properties and the amount of communication we can invest. When the dataset is balanced and IID there is negligible impact on model performance shown in Sections 4.3.1, 4.3.2 and 4.3.3. SVD indices are fairly similar since each partition of the dataset can be said to have similar variability. Therefore, each client has a similar number of communication rounds. This becomes similar to that of FederatedAveraging $\rho > 1$ since clients may have a similar number of local passes. We observe this similar behaviour for Algorithm 4 as well as DynAvg techniques. Using SVD to invest the communication becomes particularly interesting when we consider unbalanced datasets. Different partition sizes demonstrated by Figure 3.1b have different underlying structural properties. We observed that communication metrics do not differ significantly for unbalanced datasets for Algorithm 3. However, we observed this increase for DynAvg $\Delta \in \{0.3, 0.8, 2.2, 2.8\}$. In addition, DynAvg techniques requires hyperparameter tuning which introduces unnecessary complexity. It is interesting that Algorithm 3 appears to solve the problem in a different way to Algorithm 4 and DynAvg. Algorithm 3 invests more communication in the clients that have richer datasets i.e. datasets that lose less information when calculating 95% variance of the data. However, DynAvg and Algorithm 4 aim to invest communication in clients that achieve a higher loss or diverge further than a reference model. This is similar to Gradient Boosting (Friedman, 2000). Despite the differences in approaches, the results are fairly similar for IID and unbalanced datasets.

We can then stipulate that dynamic averaging with SVD and dynamic averaging based on the loss hold for IID and unbalanced datasets. However, when considering non-IID, we observed an overall decrease for all algorithms. Addressing this property requires additional measures. SVD inspects how much information is lost if we were to calculate 95% of the variance of the data. Communication is to be invested in clients with the least information lost. It does not take into account the class labels such as oversampling techniques. Nonetheless, for non-IID and unbalanced datasets, dynamic averaging with SVD shows improved performance over FederatedAveraging for number of local passes $\rho \in \{10, 20, 50, 100\}$.

Overall, both Algorithms 3 and 4 are promising frameworks for federated learning. We achieve well invested communication efficiently that has negligible impact on performance for unbalanced datasets. We did this by exploring properties of the underlying dataset as well in a federated learning setting. There are many contributions in machine learning that have utilized SVD. These include matrix factorization for recommender systems and principal component analysis (PCA). We find that in a federated learning setting, we are able to utilize SVD to analyze the underlying structural properties to invest communication efficiently. Whilst the idea of using a dynamic averaging framework based on loss is promising it requires careful consideration of how to calculate the different communication intervals per client. We observe that if the communication protocol is too dynamic, model performance degrades. With further improvements which we discussed in this section, it may be possible to achieve more stable results. We also take away that there is an optimal amount of communication that can be invested efficiently. This is directly correlated to the model performance for the different partitioning strategies. Too little communication derails gradient descent from its trajectory and thus decreases model performance. Too much communication creates communication overhead in a federated learning environment even though model performance is not impacted.

Limitations. The main aim of this study is to balance communication and computation in a federated learning environment by investing the communication efficiently. With any methods in federated learning there are limitations. Both dynamic averaging algorithms presented in Section 3.3 show promise. However, Algorithm 4 does not hold for non-IID cases. Non-IID cases are particular challenging to solve since clients may only have certain class labels. We cannot use data exchanging methods (Verma et al., 2019) as this is not well suited in a federated learning environment due to privacy risks. Algorithm 3 holds well for MNIST and Fashion-MNIST. It also remains to be seen if these frameworks hold well for tabular data. The limitation of the current architecture has only been tested up to hundreds of clients, but the patterns and algorithms have been designed to be scalable. Moreover, the infrastructure used for the experiments where based on a SLURM cluster. Such computational environments provide stable connectivity and uninterrupted experiment execution. However, as discussed in Section 2.4, this is not the case as client's have poor network connections or limited computational resources. A possible future direction would be to run experiments on geographically distributed environments which are provided by platforms such as Tensorflow Federated and OpenMined. Lastly, we have not considered quantization as apart of this study which will also

assist in reducing communication costs. Encoding and reconstructing parameters may also prove useful as a differential privacy technique.

Chapter 5

Conclusion and future work

In Chapter 1, we mentioned that the aim of this study was to contribute to dynamic averaging methods by improving how communication is invested. We used different models ranging from logistic regression to multi-layer feedforward neural networks as a means of demonstrating the efficacy of this framework. We used popular datasets, namely MNIST and Fashion-MNIST to solve a standard image classification problems. We also asked ourselves two questions. The first one being, can we use the underlying properties of local datasets to invest communication efficiently for federated learning with negligible impact on model performance? The other being, can we invest communication efficiently based on the loss of each client with negligible impact on model performance? Answering these questions are not straightforward, given the results presented in Chapter 4. In Chapter 4 we observed that solving for non-IID cases is not trivial. Both Algorithm 3 and Algorithm 4 suffer for non-IID cases across different datasets and optimizers. However, this trend is evident for popular methods such as FederatedAveraging and DynAvg. This suggests that oversampling techniques or data exchange methods are required to assist these algorithms. So, the answer to the questions posed in this study is that Algorithms 3 and 4 do not hold for non-IID cases. On the other hand, both Algorithm 3 and Algorithm 4 hold for unbalanced datasets when they are identically and independently distributed across different optimizers. Therefore, the take away is that there is direct correlation with the underlying properties and the amount of communication we can invest. The results for Algorithm 4 appear promising with minimal hyperparameter tuning. However, it requires careful consideration of how to calculate the different communication intervals per client. This is exaggerated for non-IID cases where we see Algorithm 4 suffer most in terms of model performance. An important thing that we raised in Chapter 4 is that there is no additional hyperparameters that add additional complexity such as with DynAvg. We see that when we do not carefully consider the violation threshold, we communicate as often as FederatedAveraging $\rho = 1$. This is infeasible in a federated learning setting. This is one of the major benefits of both Algorithm 3 and 4 over existing frameworks. We see that

Algorithm 3 is comparable to DynAvg and is an improvement over FederatedAveraging $\rho > 1$ for non-IID cases. Algorithm 4 is more comparable to FederatedAveraging $\rho > 1$ for non-IID cases and has a greater reduced model performance than Algorithm 3. If we consider oversampling methods as well as controlling the dynamism of Algorithm 4 it is possible to improve model performance for non-IID cases. This will lead to both Algorithm 3 and 4 holding for non-IID cases. We also provided an analysis of the SVD computation for Algorithm 3 and showed that this computation happens in $\mathcal{O}(2mn^2 - 2n^3)$ for $m \gg n$ or $\mathcal{O}(4mn^2 - \frac{4}{3}n^3)$ otherwise. However, it remains to be seen if this computation is possible on devices which have less computation power and in memory capacity. We look to investigate alternative SVD implementations in a federated learning setting as future work.

In summary, Chapter 4 highlights that successful federated learning algorithms need to focus on the key properties of federated learning. We are able to say that we can use the underlying properties of local datasets to invest communication efficiently for unbalanced datasets when they are identically and independently distributed. Similarly, we can say that we can invest communication efficiently based on the loss of each client for unbalanced datasets when they are identically and independently distributed. However, this does not hold for non-IID cases which is the case for FederatedAveraging and DynAvg. The results show promise under most cases but do have limitations in general cases. This ignites the need to continuously address the problems that exist in federated learning. The dynamic averaging with SVD strategy needs to be improved on in future work by including techniques such as oversampling to address all general cases for different types of data. In addition, different SVD implementations are to be considered in a federated learning setting. The dynamic averaging with loss strategy needs to be less dynamic and also possibly include oversampling as part of it's algorithm to address all general cases. There are also other avenues to address communication efficiency which we briefly mentioned in this study, quantization being one of them. This will further improve the communication efficiency of the dynamic averaging algorithms without impacting the accuracy of the algorithm. We also need to consider differential privacy due to model-inversion attacks in a federated learning setting. By increasing the complexity of these models as well as adding techniques to improve model performance, we may implicitly be addressing differential privacy as discussed in Section 2.3. Differential privacy may increase the amount of computation that is needed and therefore provides some exciting challenges. The details of this computation is provided in Section 2.3. Moreover, considering a client as a server would pose interesting challenges and benefits as well as dynamic server appointment. Unfortunately, the framework was not implemented on Android, and we look to

implement this framework on Android given that ZeroMQ is available in multiple languages including Java. Overall, this study has addressed the key aspects of federated learning and ways to improve them. Despite the limitations, there is a way forward to further improve dynamic averaging techniques in federated learning.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I. J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D. G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P. A., Vanhoucke, V., Vasudevan, V., Viégas, F. B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., & Zheng, X. (2016a). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467.

Abadi, M., Chu, A., Goodfellow, I., McMahan, H. B., Mironov, I., Talwar, K., & Zhang, L. (2016b). Deep learning with differential privacy. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*.

Alistarh, D., Li, J., Tomioka, R., & Vojnovic, M. (2016). QSGD: randomized quantization for communication-optimal stochastic gradient descent. *CoRR*, abs/1610.02132.

Alpaydin, E. (2020). *Introduction to Machine Learning*. The MIT Press, 4th edition.

Blaise Barney, L. C. (2008). Introduction to parallel computing. `https://computing.llnl.gov/tutorials/parallel_comp/`.

Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H. B., Patel, S., Ramage, D., Segal, A., & Seth, K. (2017). Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17 (pp. 1175–1191). New York, NY, USA: Association for Computing Machinery.

Bottou, L., Curtis, F. E., & Nocedal, J. (2016). Optimization Methods for Large-Scale Machine Learning. *ArXiv e-prints*.

Brendan McMahan and Daniel Ramage (2017). Federated learning: Collaborative machine learning without centralized training data. `https://ai.googleblog.com/2017/04/federated-learning-collaborative.html`. [Online; accessed 12-November-2019].

Briggs, C., Fan, Z., & Andras, P. (2020). Federated learning with hierarchical clustering of local updates to improve training on non-iid data.

Chen, W., Wang, Z., & Zhou, J. (2014). Large-scale l-bfgs using mapreduce. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 27* (pp. 1332–1340). Curran Associates, Inc.

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., & Ng, A. Y. (2012). Large scale distributed deep networks. In *NIPS*.

Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Tomov, S., & Yamazaki, I. (2018). The singular value decomposition: Anatomy of optimizing an algorithm for extreme scale. *SIAM Review*, 60(4), 808–865.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12, 2121–2159.

Fredrikson, M., Jha, S., & Ristenpart, T. (2015). Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15 (pp. 1322–1333). New York, NY, USA: Association for Computing Machinery.

Friedman, J. H. (2000). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29, 1189–1232.

Geyer, R. C., Klein, T., & Nabi, M. (2017). Differentially private federated learning: A client level perspective. *CoRR*, abs/1712.07557.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Goodfellow, I. J., Vinyals, O., & Saxe, A. M. (2014). Qualitatively characterizing neural network optimization problems.

Gopal, S. & Yang, Y. (2013). Distributed training of large-scale logistic models. In S. Dasgupta & D. McAllester (Eds.), *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research* (pp. 289–297). Atlanta, Georgia, USA: PMLR.

Hintjens, P. (2013). Zeromq: Messaging for many applications.

Hinton, G. (2012). Neural networks for machine learning - lecture 6a - overview of mini-batch gradient descent. `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

Hoefler, T. & Schneider, T. (2012). Optimization principles for collective neighborhood communications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12 (pp. 98:1–98:10). Los Alamitos, CA, USA: IEEE Computer Society Press.

Hsia, C.-Y., Zhu, Y., & Lin, C.-J. (2017). A study on trust region update rules in newton methods for large-scale linear classification. In M.-L. Zhang & Y.-K. Noh (Eds.), *Proceedings of the Ninth Asian Conference on Machine Learning*, volume 77 of *Proceedings of Machine Learning Research* (pp. 33–48).: PMLR.

Kairouz, P., McMahan, H. B., Avent, B., Bellet, A., Bennis, M., Bhagoji, A. N., Bonawitz, K., Charles, Z., Cormode, G., Cummings, R., D'Oliveira, R. G. L., Rouayheb, S. E., Evans, D., Gardner, J., Garrett, Z., Gascón, A., Ghazi, B., Gibbons, P. B., Gruteser, M., Harchaoui, Z., He, C., He, L., Huo, Z., Hutchinson, B., Hsu, J., Jaggi, M., Javidi, T., Joshi, G., Khodak, M., Konečný, J., Korolova, A., Koushanfar, F., Koyejo, S., Lepoint, T., Liu, Y., Mittal, P., Mohri, M., Nock, R., Özgür, A., Pagh, R., Raykova, M., Qi, H., Ramage, D., Raskar, R., Song, D., Song, W., Stich, S. U., Sun, Z., Suresh, A. T., Tramèr, F., Vepakomma, P., Wang, J., Xiong, L., Xu, Z., Yang, Q., Yu, F. X., Yu, H., & Zhao, S. (2019). Advances and open problems in federated learning.

Kamp, M., Adilova, L., Sicking, J., Hüger, F., Schlicht, P., Wirtz, T., & Wrobel, S. (2018). Efficient decentralized deep learning by dynamic model averaging. *CoRR*, abs/1807.03210.

Kamp, M., Boley, M., Keren, D., Schuster, A., & Sharfman, I. (2014). Communication-efficient distributed online prediction by dynamic model synchronization. In T. Calders, F. Esposito, E. Hüllermeier, & R. Meo (Eds.), *Machine Learning and Knowledge Discovery in Databases* (pp. 623–639). Berlin, Heidelberg: Springer Berlin Heidelberg.

Kingma, D. P. & Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Klenk, B. & Fröning, H. (2017). An overview of mpi characteristics of exascale proxy applications. In J. M. Kunkel, R. Yokota, P. Balaji, & D. Keyes (Eds.), *High Performance Computing* (pp. 217–236). Cham: Springer International Publishing.

Konečný, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., & Bacon, D. (2016). Federated learning: Strategies for improving communication efficiency.

Krizhevsky, A. (2009). *Learning multiple layers of features from tiny images.* Technical report.

Le Cun, Y., Jackel, L. D., Boser, B., Denker, J. S., Graf, H. P., Guyon, I., Henderson, D., Howard, R. E., & Hubbard, W. (1989). Handwritten digit recognition: applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11), 41–46.

LeCun, Y. & Cortes, C. (2010). MNIST handwritten digit database.

Li, M., Andersen, D. G., Smola, A. J., & Yu, K. (2014a). Communication efficient distributed machine learning with the parameter server. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 27* (pp. 19–27). Curran Associates, Inc.

Li, M., Anderson, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., & Su, B.-Y. (2014b). Scaling distributed machine learning with the parameter server. In *Operating Systems Design and Implementation (OSDI)* (pp. 583–598).

Lin, C.-J., Weng, R. C., & Keerthi, S. S. (2007). Trust region newton methods for large-scale logistic regression. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07 (pp. 561–568). New York, NY, USA: ACM.

Lin, C. Y., Tsai, C. H., Lee, C. P., & Lin, C. J. (2014). Large-scale logistic regression and linear support vector machines using spark. In *2014 IEEE International Conference on Big Data (Big Data)* (pp. 519–528).

McMahan, H. B., Moore, E., Ramage, D., & y Arcas, B. A. (2016). Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629.

Mostafa, H. (2019). Robust federated learning through representation matching and adaptive hyper-parameters.

Patro, S. G. K. & Sahu, K. K. (2015). Normalization: A preprocessing stage. *CoRR*, abs/1503.06462.

Recht, B., Re, C., Wright, S., & Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, & K. Q.

Weinberger (Eds.), *Advances in Neural Information Processing Systems 24* (pp. 693–701). Curran Associates, Inc.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747.

Sahu, A. K., Li, T., Sanjabi, M., Zaheer, M., Talwalkar, A., & Smith, V. (2018). On the convergence of federated optimization in heterogeneous networks. *CoRR*, abs/1812.06127.

Seide, F., Fu, H., Droppo, J., Li, G., & Yu, D. (2014). 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*.

Shokri, R. & Shmatikov, V. (2015). Privacy-preserving deep learning. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15 (pp. 1310–1321). New York, NY, USA: ACM.

Sohl-Dickstein, J., Poole, B., & Ganguli, S. (2014). Fast large-scale optimization by unifying stochastic gradient and quasi-newton methods. In E. P. Xing & T. Jebara (Eds.), *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research* (pp. 604–612). Bejing, China: PMLR.

Solihin, Y. (2015). *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall/CRC, 1st edition.

Stich, S. U. (2018). Local sgd converges fast and communicates little.

Strom, N. (2015). Scalable distributed dnn training using commodity gpu cloud computing. In *INTERSPEECH*.

Taylor, G., Burmeister, R., Xu, Z., Singh, B., Patel, A., & Goldstein, T. (2016). Training neural networks without gradients: A scalable ADMM approach. *CoRR*, abs/1605.02026.

Vepakomma, P., Swedish, T., Raskar, R., Gupta, O., & Dubey, A. (2018). No peek: A survey of private distributed deep learning. *CoRR*, abs/1812.03288.

Verma, D. C., White, G., Julier, S., Pasteris, S., Chakraborty, S., & Cirincione, G. (2019). Approaches to address the data skew problem in federated learning. In T. Pham (Ed.), *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, volume 11006 (pp. 542 – 557).: International Society for Optics and Photonics SPIE.

Vetter, J. S. & Mueller, F. (2003). Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *J. Parallel Distrib. Comput.*, 63(9), 853–865.

Woodruff, D. P. (2014). Computational advertising: Techniques for targeting relevant ads. *Foundations and Trends® in Theoretical Computer Science*, 10(1-2), 1–157.

Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747.

Xie, P., Kim, J. K., Zhou, Y., Ho, Q., Kumar, A., Yu, Y., & Xing, E. P. (2015). Distributed machine learning via sufficient factor broadcasting. *CoRR*, abs/1511.08486.

Xing, E. P., Ho, Q., Xie, P., & Wei, D. (2016). Strategies and principles of distributed machine learning on big data. *Engineering*, 2(2), 179 – 195.

Yang, Q., Liu, Y., Chen, T., & Tong, Y. (2019). Federated machine learning: Concept and applications. *CoRR*, abs/1902.04885.

Zhang, S., Choromanska, A., & LeCun, Y. (2014). Deep learning with elastic averaging SGD. *CoRR*, abs/1412.6651.

Zhang, Y. & Xiao, L. (2015). Disco: Distributed optimization for self-concordant empirical loss. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15 (pp. 362–370).: JMLR.org.

Zhao, Y., Li, M., Lai, L., Suda, N., Civin, D., & Chandra, V. (2018). Federated learning with non-iid data. *CoRR*, abs/1806.00582.

Zhuang, Y., Chin, W.-S., Juan, Y.-C., & Lin, C.-J. (2015). Distributed newton methods for regularized logistic regression. In T. Cao, E.-P. Lim, Z.-H. Zhou, T.-B. Ho, D. Cheung, & H. Motoda (Eds.), *Advances in Knowledge Discovery and Data Mining* (pp. 690–703). Cham: Springer International Publishing.